Dynamic Simulation of Manipulation & Assembly Actions

Thulesen, Thomas Nicky

Publication date:
2016

Document Version
Peer reviewed version

Dynamic Simulation of Manipulation & Assembly Actions

Thomas Nicky Thulesen
The Maersk Mc-Kinney Moller Institute
Faculty of Engineering
University of Southern Denmark

PhD Dissertation

Odense, November 2015

The Maersk Mc-Kinney Moller Institute
Faculty of Engineering
University of Southern Denmark

Campusvej 55
5230 Odense M, Denmark
Phone +45 6550 3541
www.mmmi.sdu.dk

Abstract

To grasp and assemble objects is something that is known as a difficult task to do reliably in a robot system. Yet humans are able to quickly learn how to manipulate and assemble objects based on previous experience. Most learn to do these tasks as infants and the knowledge from previous experience allows solution of new similar, but different, tasks intuitively. In this dissertation dynamic simulation is seen as a necessary tool to allow robots to get the same intuition for solving manipulation and assembly tasks. By virtual modelling of the task, simulation will allow robots to reason about the optimum strategies without complex programming by highly skilled specialists.

In this dissertation, a different paradigm is proposed for performing dynamic simulation in the context of tight-fitting assembly. A new physics engine is developed for this purpose. It focuses very much on the geometric aspects of contact generation and temporal coherence between contacts. This allows enhanced possibilities regarding modelling of the interaction between bodies in contact. Regarding development of robust control strategies for assembly, it is important to have a simulated environment that is realistic compared to the real environment. The problem with simulation of tactile sensors is that they lead to redundancy in the mathematical formulation of the dynamical system. In this dissertation, we present a method that includes an objective for distributed force measurements in the case of redundancy, and we show that this objective causes simulated force measurements that are more realistic and more ideal for control strategies.

The main purpose of the work performed in this dissertation is to make an engine which is easily extendable, well-documented and also has interfaces that makes it transparent what happens step-by-step during execution. An assembly simulation framework is presented with the purpose to create a unified way of defining assembly actions and results. This also forces the user to write control strategies that are reusable in both simulation and the real world.

Different comparisons are performed in order to qualitatively illustrate the benefits from using the newly developed engine for manipulation and assembly. This shows that our approach to dynamic simulation is viable and that, even though it adds some complexity to contact management, it is still very efficient in the tight-fitting assembly use case, while at the same time allowing detailed modelling of the interaction.

Resumé

In Danish

Et kendt problem inden for robotteknologi er at gribe og samle objekter på en pålidelig måde. For mennesker er håndtering og samling af objekter en evne, som hurtigt kan tilegnes baseret på forudgående erfaringer. Disse evner tilegnes typisk i børneårene, og på baggrund af vores erfaringer bliver løsning af nye, lignende opgaver intuitiv. I denne afhandling ses dynamisk simulering som et nødvendigt værktøj for at give robotter en lignende intuition for at gribe, håndtere og samle objekter. Ved at opstille en virtuel model vil simulering gøre det muligt for robotten selv at ræsonnere sig frem til den optimale strategi, fremfor at denne skal programmeres eksplicit af højtuddannede specialister.

I denne afhandling foreslås et nyt paradigme til dynamisk simulering af samling af objekter med lav tolerance. Vi har udviklet en ny fysikmotor med særlig fokus på modellering af kontinuerlig kontakt mellem legemer. Dette giver nye muligheder for at anvende detaljerede modeller for interaktion mellem legemer, der er i kontakt. Udvikling af robuste kontaktstrategier til at samle objekter kræver et simuleringsmiljø, som er realistisk i sammenligning med et fysisk miljø. Simulering af taktile sensorer kan føre til redundans i den matematiske formulering af det fysiske system. Her præsenteres en simuleringsmetode, som involverer distribuerede kræfter som et objektiv i tilfælde af redundans. Det illustreres, hvordan et sådan objektiv giver simulerede kraftmålinger, som er mere realistiske og mere ideelle for udvikling af kontrolstrategier.

Hovedformålet med det udførte arbejde er at udvikle en fysikmotor, der kan udvides og er veldokumenteret samt har grænseflader, der gør det transparent, hvad der sker skridt for skridt under simulering. Et simuleringsframework for samling af objekter præsenteres med det formål at udvikle et generelt format for definitionen af samleopgaver og resultater. Målet er, at brugeren tvinges til at skrive kontrolstrategier, som er generiske og kan genbruges både i simulering og i en fysisk opstilling.

Forskellige sammenligninger er udført for kvalitativt at illustrere fordelene ved at bruge den nyudviklede fysikmotor til håndtering og samling af objekter. Derved redegøres for, at metoden er en lovende tilgang til dynamisk simulering, samt at fysikmotoren på trods af øget kompleksitet ved detektering af kontakter stadig er effektiv til samling af objekter. Samtidig er det muligt at definere detaljerede modeller for interaktionen mellem objekter.

Preface & Acknowledgements

For the past four years, I have worked exclusively with dynamic simulation in robotics, focusing on some of the hard fundamental issues in the field. The work performed in my PhD is a continuation of my master's thesis [1], in which I presented some basic theory about dynamic grasp simulation in the context of grasping of objects. For readers not familiar with advanced rigid body dynamics, the thesis provides a good starting point. It outlines many of the complex issues that are addressed in this dissertation. It is assumed that the reader is familiar with physics, mathematics and robotics at a graduate level. Otherwise, the dissertation will be difficult to follow.

The project has been funded by the national projects patient@home and Center for Advanced Robotics Manufacturing Engineering (CARMEN). In the former project the context is using service robots in healthcare by assisting patients in their homes after being discharged from hospital. In CARMEN, the task is to perform manipulation actions in an industrial context. Manipulating objects using robots is a fundamental challenge regardless of whether manipulation is being used in service robotics or in industrial applications.

I want to thank Professor Henrik Gordon Petersen for supervision and guidance during the years. Much of the work presented in this dissertation are based on a common software framework used in the group, and I want to thank Jimmy Alison Rytz and Lars-Peter Ellekilde for their previous work on the framework.

I would like to thank former and present members of the Cognitive and Applied Robotics group at the University of Southern Denmark. People that I have worked with and come to know during the years. This includes Lilita Kiforenko, Leon Bodenhagen, Nadezda Rukavishnikova, Preben Hagh Strunge Holm, Professor Norbert Krüger, Thiusius Rajeeth Savarimuthu, Dirk Kraft, Anders Glent Buch, Jacob Pørksen Buch, Thomas Fridolin Iversen, Thorbjørn Mosekjær Iversen, Troels Bo Jørgensen, Johan Sund Laursen, Simon Mathiesen, Ole Wennerberg Nielsen, Stefan-Daniel Suvei, Lars Carøe Sørensen, Mikkel Tang Thomsen, Zhiyuan Xie, Michael Bejer Andersen, Dorthe Sølvason, Adam Wolniakowski and Severin Fichtl.

Finally, I would like to thank my parents, Thule and Jonna, my sister, Michaela Louise Thulesen, and my friends for their patience, understanding and support during the years. Also thanks to Michaela for proofreading.

Contents

Figures

Tables

Algorithms

Listings

Notation & Terminology

Notation

Throughout the thesis, some common mathematical notations will be used to make the math more readable.

$\mathbf{A}$	Matrices are written in bold font with uppercase letters.
$\mathbf{b}$	Vectors are written in bold font with lowercase letters.
$\mathbb{S}$	Sets are in open face capital letters.
M, Σ	Abstract entities, maps and tuples are in uppercase.
s, ϵ	Scalars are in lowercase.
$\mathfrak{R}$	Vector spaces are in fraktur font.
$\mathbf{v}^\times$	Short notation for a skew-symmetric matrix representing the cross product. $\mathbf{v} \times \mathbf{b}$ is equivalent to $\mathbf{v}^\times \mathbf{b}$, where $\mathbf{v}^\times = \begin{bmatrix} 0 & -v_3 & v_2 \\ v3 & 0 & -v1 \\ -v_2 & v_1 & 0 \end{bmatrix}$
MET	Method names are written in small caps.

Furthermore, the following reserved entities are used:

$\mathbf{I}$	An inertia matrix.
$\mathbf{J}$	An identity matrix.

The hat notation, $\widehat{\mathbf{A}}$, refers to entities that have been transformed into a constraint subspace. The concept is introduced in section 3.5.1.

Terminology

Certain terminology and shortcuts are used throughout the thesis:

BCG	The Body Constraint Graph which is the topic of section 3.1.
DAE	Differential Algebraic Equation
DCP	Differential Complementarity Problem
EAA	Equivalent Angle Axis
LCP	Linear Complementarity Problem
MB	Megabyte (10^6 bytes)
ODE	Open Dynamics Engine (*not* Ordinary Differential Equations)
PAL	Physics Abstraction Layer which is described in section 2.1.1
PD	Proportional Derivative (a type of controller)
PSD	Used to refer to a Positive Symmetric Definite matrix.
QP	Quadratic Programming
RWPE	The RobWorkPhysicsEngine. The engine is introduced in chapter 3.
SVD	Singular Value Decomposition

CHAPTER 1

Introduction

Simulation has become a widely used tool within robotics. It allows inexpensive design and optimisation by using a virtual model of the robot and its surroundings, instead of doing the implementation directly on physical robot systems. This way, new algorithms can be tested in a safe environment without breaking the physical robot and avoiding hazards in the environment. For complex robot systems, the simulator must be able to realistically simulate sensors and actuators used in the corresponding physical system, including the noise and uncertainties that can be expected. The simulator allows development of robust control algorithms that can be applied in reality with equal performance. Earlier, robots have been used for automation in large-scale production, but today they are needed for changing production environments and in the area of service robotics as well.

In this chapter, the use of simulation in robotics is introduced. Simulation is a broad term which in a robotics context usually refers to kinematic simulation. Kinematic and dynamic simulation are introduced along with a discussion of the main uses of each, before considering an example of a manipulation task and the requirements and challenges, manipulation poses for simulators. Physics engines perform the required computations for dynamic simulations and this concept will be introduced. A brief introduction to the RobWork framework is given, and finally the main contributions of this dissertation will be considered along with an overview of the remainder of the dissertation.

1.1 Kinematic Simulation

In kinematic simulation, the motion of a system of bodies is simulated without considering which forces or torques that cause these motions. In robotics, this is for instance used to model robot arms, where multiple bodies are connected in serial by joints that allow relative rotation. In this case, forward kinematics can be used to determine the position of the bodies, given the relative angles, or inverse kinematics can be used to do the opposite, namely determining the angles, which cause the end of the arm to reach a certain point in space. Besides considering positions, the velocities and accelerations can also be studied in kinematic simulation. Note that in kinematic simulation objects do not have weight and they feel no forces. Usually, geometry is attached to the bodies for visualisation purposes and for use in algorithms, which plan collision free kinematic motion.

If it can be assumed that the motors in a physical robot are powerful enough, kinematic simulation is a great tool for planning motions as it can be expected that the robot will

provide the required torque to reach a given position. For industrial robots, this will always be the case, but as robots get smaller, lighter and cheaper, limitations of motors should also be addressed in simulation.

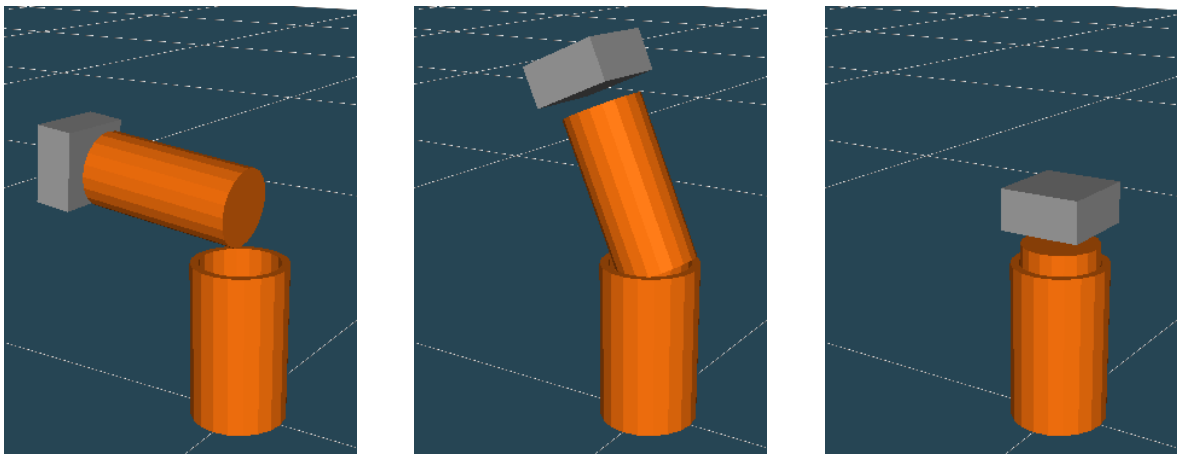
The major limitation of kinematic simulation is, however, its inability to simulate manipulation of objects. Opposite to the robot arm, where the degrees of freedom are limited and under full control, manipulation of objects can have many degrees of freedom, that are not controlled. To get a realistic behaviour of these objects, the physical laws of motion must be used. In this case, contact forces and friction modelling become important for determining the interactions and motions of the objects. Unfortunately, this is impossible using only kinematic simulation.

1.2 Dynamic Simulation

Dynamic simulation is the task of determining body motion from the forces and torques acting on a given body. Inverse dynamics does the opposite, namely tries to determine what forces and torques must act on the bodies to achieve a certain motion. Hence, dynamic simulation will determine torques for each joint of a robot arm in order to keep the joints together and moving as required. For manipulation actions, dynamic simulation allows modelling of contact forces and friction making the simulation more realistic. The drawback of dynamic simulation is that it will typically require more computational power.

1.3 Simulation of Manipulation & Assembly

In figure 1.1, an example is shown of an assembly procedure. Here the task is to simply insert the cylinder into the tube. The cylinder is attached to the grey box, hence the position of the cylinder is controlled by moving the box. Both the cylinder and tube are free to move and the tube is initially resting on a fixed plane. As there are only two dynamic bodies in the system, it is a fairly simple dynamical system. If contacts are modelled correctly, there will also be few contacts. Many available simulation tools are based on dynamics developed for simulation in games and animation, where there can be many bodies in contact and the dynamics must be solved in real-time. In our case, it is however possible to take a slightly different approach as we will expect the number of bodies in simulation of manipulation actions fairly limited. This means that more time can be spent on computing correct and realistic dynamics instead of focusing on hard real-time requirements. As will become apparent later, contact detection for an assembly scenario, like the one shown, will often generate a lot of contacts in the tight-fitting scenarios. Many algorithms do not handle penetrations well and require contact layers. In a tight-fitting scenario like this, the contact layer must be small and hence the time-steps must be small. Control of the cylinder is in this case done with a spring, introducing compliance to the system. Care must be taken as stiff springs are a challenge when doing simulation, often requiring small time-steps to be taken. Simulations of manipulation actions



(a) Initial configuration (b) Insertion (c) Final configuration

Figure 1.1: Example of a circular peg-in-hole assembly sequence.

often result in redundant configurations, in which multiple solutions are possible. This can be a problem if the solution should be used in control strategies for instance, in which case one must be careful with what data is retrieved from the simulator. Typically, simulators do not care about force-space results, but only if the motion, that is produced, seems realistic.

1.4 Physics Engines

The component in a dynamic simulator performing the core dynamical computations is typically called a *Physics Engine*. Here the focus lies on rigid body engines, but some engines also provide other physics such as soft body dynamics, cloth or fluid dynamics. Selecting an engine depends very much on the use case, but parameters often considered are:

Open-Source & Licenses As physics engines are often used as part of larger frameworks, it is important to consider which license the engine distributes with. Typically, physics engines are open-source, while the frameworks, that use them, might not be.

Types of Physics For instance fluids, soft bodies and rigid bodies can be relevant.

Constraint Types If used in robotics it is important that different types of joints are available. Sometimes a limited set of joints can force the user to create dummy bodies as a workaround, which is of course not ideal.

Motors & Limits Often, it is desirable to control robot arms by setting the velocity of joints directly.

Sensors In some cases, force/torque and tactile sensors need to be simulated.

Contacts Models Contacts modelled as being soft might not always be ideal. Depending on whether penetrations can be tolerated or not.

Contact Detection The primitives supported by the engine are important as they can limit the freedom of modelling the bodies.

Springs/Compliance For modelling of compliant devices it is important to have the ability to model springs. Some engines support springs indirectly by tweaking some internal parameters, while others have explicit support for specification of springs.

Material Modelling If the engine must handle contacts, modelling of friction is known to be a difficult issue. Therefore, the supported friction models are typically very simple, if any at all.

Speed Often speed is important, but as speed and accuracy are often a trade-off, it might in some applications be better with higher accuracy and realism.

1.5 The RobWork Framework

The work presented in this dissertation has been integrated into the RobWork framework [2]. RobWork consists of four main components, namely RobWork, RobWorkStudio, RobWorkSim and RobWorkHardware. RobWork is the core library providing common tools for mathematics, scene definitions, interchange formats, path planning and similar. In figure 1.2

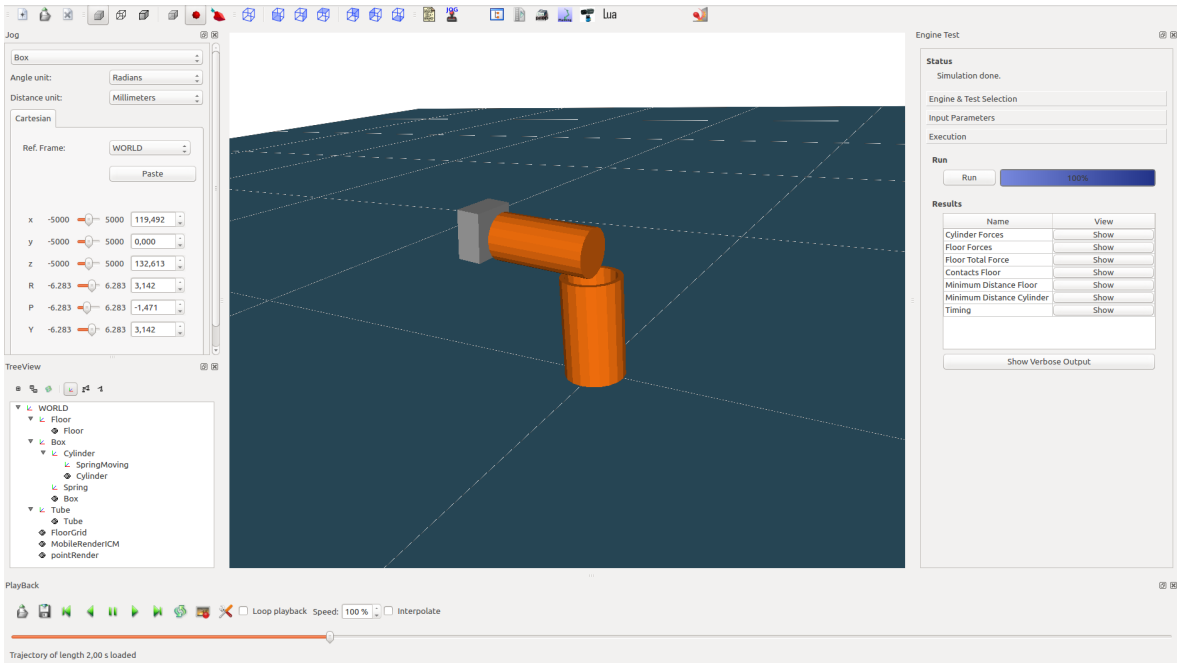


Figure 1.2: Screenshot from simulation in RobWorkStudio.

the RobWorkStudio application is shown. It is the graphical user interface for the RobWork framework. RobWorkHardware provides links between RobWork and physical robots and grippers. Finally, RobWorkSim is a package on top of RobWork that adds dynamic simulation features. This can be used independently or through a plugin to RobWorkStudio as

shown on the right in the figure. In the bottom of the window, it is possible to playback a simulation, whether it is kinematic or dynamic, and on the left it is possible to move frames and visualise different frames.

1.6 Contributions

The major contribution of this work is a new open source physics engine named RobWork-PhysicsEngine (RWPE). The engine is targeted simulation in robotics with a particular focus on manipulation actions with the following key features:

- The developed engine is based on iterative methods that gives the user the ability to balance speed and accuracy as well as it gives stable and realistic forces, which is usable in control strategies.
- Accurate integration of motion with a second-order method allowing modelling of stiffer springs and low-compliance devices.
- Event-based time-stepping where collisions are handled at the exact time of impact.
- Hard contacts and penetration avoidance by using correction by projection instead of stabilisation methods like Baumgarte.
- A micro-slip friction model, which models hysteresis behaviour and gives smooth transitions instead of the Coulomb model. The Coulomb model is usually approximated anyway.
- In the domain of collision detection, a new contact tracking approach is presented. This will allow having stateful contacts, which we believe are of vital importance for accurate simulation in the sense that penetrations can be avoided, higher-order integration can be used and much better friction models can be utilised.

Along with this, a modular software framework has been designed, including the following features:

- Easy extension with user-defined models is facilitated. This, for instance, includes friction, restitution models and integration schemes.
- Advanced debugging and test tools with detailed logging and associated visualisation tools have been developed.
- A complete framework is presented for simulation of atomic assembly actions. It makes it possible to develop control strategies that are equally applicably in simulation as in real life.
- To make the engine able to adapt to the resources given, a framework for parallelisation is proposed.
- An interface for MatLab is provided. It allows development of control strategies in this environment.
- Integration of Bullet into an unified engine interface in RobWorkSim, such that ODE, Bullet and RWPE are now supported.

A final contribution is a comparison of RWPE with state-of-the-art engines like Open Dynamics Engine (ODE) and Bullet. The RWPE engine is also presented in an article [3] submitted to *International Conference on Robotics and Automatic* (ICRA).

1.7 Overview

In chapter 2, an overview of existing simulation tools is given along with a discussion of state of the art in physics engines for rigid body dynamics. RWPE is presented in chapter 3 and a dedicated chapter 4 discusses choice of engine parameters and the implications. A comparison between RWPE and other engines is given in chapter 5, and in chapter 6, considerations on the software design is given along with examples of how the modular design allows users to write their own modules for the engine. In chapter 7, the framework for assembly simulation will be presented along with results of using different engines for assembly simulation. Finally, a conclusion will be given in chapter 8, and future work will be discussed in chapter 9.

CHAPTER 2

State of the Art

We will look into the existing simulation software that exists for use in mechanics and robotics. A distinction is made between these software packages that are typically intended for use directly by the user, and the physics engines used for doing the actual dynamics. A physics engine will typically not have a user interface on its own, but is provided as a component that can be used by these higher level software packages. After looking into the existing software, state of the art in multi-body dynamics is presented followed by a discussion of the relation to the existing engines. Finally modelling of bodies and their geometries is considered, along with state of the art collision detection methods.

2.1 Overview of High-level Software for Robot Simulation

In table 2.1 an overview of different proprietary robot simulators is given. The industrial robot manufacturers, ABB, Adept, Denso, FANUC, Kawasaki and KUKA each have their own simulation environments. The advantage of using these is that the physical robot controllers are modelled much better by the manufacturer, as it is often unknown to others how they work. They are typically promoted for classical applications in industrial robots, like pick and place, welding, palletizing and material removal. The main selling point for these packages is fast and cheap offline robot programming that is easily transferred to the physical robots in native language. None of these manufacturers have explicitly promoted features for dynamic simulation. In the table an overview of some independent simulation products is given as

Robot Manufacturers
ABB RobotStudio [4]
Adept ACE [5]
Denso WINCAPS [6]
FANUC RoboGuide [7]
Kawasaki K-ROSET [8]
KUKA.Sim [9]

Software Companies	
3DSimulate & 3DRealize [10]	
Actin [11]	RoboDK [12]
anyKode Marilou [13]	RoboLogix [14]
Artiminds Robotics [15]	RoboWorks [16]
DELMIA [17]	Webots [18]
MSC ADAMS [19]	WorkcellSimulator [20]
Microsoft Robotics Developer Studio [21]	

Table 2.1: Different examples of proprietary robot simulators.

well. These are provided mostly by companies that deliver software only. These can be expected to support a wider range of robots, and allows creation of virtual robot cells, similar to the ones possible in software from the robot manufacturers. Still many do not support dynamic simulation. Marilou, Microsoft Robotics Developer Studio (MRDS) and Webots

all reveal that they rely on external engines for dynamic simulation. While Marilou and Webots rely on the Open Dynamics Engine, MRDS can use PhysX. Actin and Adams have dynamic simulation, but the details of implementation are unknown. The main drawback of proprietary software is that it is in difficult to know how the dynamics is implemented. The main target of Webots is mobile robotics, and the main focus of Adams is mechanics and structural analysis in general. ArtiMinds specialises in fast and intuitive programming of complex robot tasks, such as peg-in-hole assemblies using force/torque sensors. For this purpose it has 29 template libraries for force-adaptive tasks. Of the commercial software available, ArtiMinds is clearly the tool that comes closest to our need of dynamic simulation for manipulation actions.

In table 2.2 an overview of different higher-level open-source simulators for robotics are shown. Higher-level refers to the fact that these are frameworks that uses different underlying physics engines to do the actual dynamics. This distinction between the core physics engine and the higher-level framework on top is however not always clear as these can be more or less integrated into each other. The individual engines will be discussed in section 2.3. The

Name	**Language**	**3D Physics Engines**	**Target**
ARGoS [22]	C++	ODE, Custom Particle Engine	Multi-engine Swarm Robotics
ARS [23]	Python	ODE	Multi-engine Swarm Robotics
Dance [24]	C++	ODE, SD/FAST	Biomechanics, Animation and Control
Gazebo [25]	C++	ODE, Bullet, Simbody, DART	Mobile Robots
OpenGRASP [26, 27]	C++, Python	PAL/FISICAS	Grasping and Manipulation
OpenHRP [28]	C++	LCP, Featherstone	Humanoids
OpenRAVE [29]	C++	ODE, Bullet	Robot Manipulation
OpenSim [30]	C++	SD/FAST, Simbody	Biomechanics
MORSE [31]	Python	Bullet	Mobile Robots
RobWorkSim	C++	ODE, Bullet	Robot manipulation
SimSpark [32]	C++, Ruby	ODE	Multi-agent Simulation
Torque3D	C++	PhysX, Bullet, Built-in	Games, Animation
V-Rep EDU [33]	C++	ODE, Bullet, Newton, Vortex	Robotics

Table 2.2: Open-source robot simulators.

table clearly shows that many of the found open-source frameworks are written in C++, and only a couple does not support ODE out of the box. ODE is clearly the engine of choice in many frameworks, but many provide a wider array of different choices. The frameworks can be categorised into four overlapping categories, namely games and animation, mobile and multi-agent systems, biomechanics and manipulation. This categorisation is based on how the authors present the software themselves.

One of the main objectives for simulation of multi-agent systems is the possibility of running many independent dynamic simulations in parallel. *ARGoS* is a multi-physics robot simulator for simulation of swarm robotics, with focus on real-time simulation of large swarms of heterogeneous robots. It has an objective of allowing assignment of different engines to different parts of the environment. *Autonomous Robot Simulator (ARS)* is a Python engine for simulation of mobile manipulators. *Modular Open Robots Simulation Engine (MORSE)* focus on realistic simulation of indoor and outdoor environments with autonomous robots. *SimSpark* is an engine for multi-agent simulations for AI and robotics research, and *Gazebo* is an engine that is often used together with the Robot Operating System (ROS) [34] for simulation of primarily mobile robots. Common to all of these simulators is that they typically rely on either the ODE or Bullet engine for doing dynamics, with the exception of Gazebo which support Simbody and DART as well. In focus is simulation and planning for mobile robots and simulation of multiple independent platforms simultaneously.

In biomechanics simulators should be able to handle articulated bodies with many bodies and joints reliably. *Dynamic Animation and Control Environment (DANCE)* is developed for articulated structures and targets both biomechanics applications and computer animation. The *Open Architecture Humanoid Robotics Platform (OpenHRP)* is a simulator which allows development of controllers for humanoids, for instance for stable walking. Finally *OpenSim* is a software framework that allows development of models of musculoskeletal structures. The objective is development of controllers, analysis and advanced modelling of both contacts and muscles. These simulators clearly have certain requirements that need different underlying engines for dynamics. Here engines like SD/FAST and Simbody is used, which uses certain articulated dynamics. OpenHRP uses its own internal methods which has similar objectives. In section 2.2 this will be discussed in more detail.

Simulation of manipulation has slightly different objectives than in biomechanics, as there are typically many degrees of freedom that is uncontrolled. *OpenGRASP* is an engine for grasp simulation, and is actually an extension to the *Open Robotics Automation Virtual Environment (OpenRAVE)* framework. Similar to our own *RobWorkSim*, it is a modular extendible framework with interfaces that allows integration of different engines. Both OpenRAVE and RobWork is mainly used for simulation of robots and automation in an industrial context. OpenGRASP is a plugin that provides grasp simulation functionality on top of OpenRAVE. Grasp simulation focus on the reliable grasping of objects with different types of dexterous grippers.

To summarise, many simulators exist for robotics and typically these falls into certain areas of focus. Many of the proprietary ones are kinematic simulators only, but many open simulators exists which incorporates dynamic simulation. In the end most of the higher-level simulators rely on the same underlying physics engine for doing dynamics. In particular ODE and Bullet seems to be common choices. A recent survey [35] confirms that especially in the robotics research community, ODE and Bullet is the major engines of choice for dynamic

simulations. 119 researchers were asked about which tools they use, and the most popular ones seem to be Gazebo, ARGoS, ODE, Bullet, V-Rep, Webots and OpenRave. Note that most of these are simply higher level frameworks that in the end use ODE or Bullet as the underlying engine.

2.1.1 Abstraction Layers & Data Interchange

Using an abstraction layer makes it possible to easily use a variety of different engines that conforms to the interface. The abstraction layers can be seen as a middle layer in-between the high-level frameworks already discussed and the low-level physics engines which will be discussed in section 2.3.

An example of this is the *Physics Abstraction Layer (PAL)* [36, 37], which is a C++ layer with support for ODE, Bullet, JigLib, Newton, PhysX, Tokamak and TrueAxis. Furthermore experimental support includes Havok, IBDS, OpenTissue and Simple Physics Engine. The Open Physics Abstraction Layer [38] has a similar purpose but supports only ODE. It is no longer developed and suggests PAL as an alternative.

OpenGRASP uses their own abstraction *FISICAS* as they did not find PAL satisfactory [27]. The critique is that PAL has a chaotic design, is unstable and is difficult to extend. The FISICAS layer allows using Bullet and ODE. In RobWorkSim a unified engine layer was developed based on specific requirements such as the need for simulation of tactile array sensors and cameras [39]. The target here is to make a higher-level abstraction than in PAL, allowing the user to implement custom sensors and controllers. The unified physics engine interface in RobWorkSim was presented with the ability to use ODE, Bullet and RobWorkPhysicsEngine (RWPE). In practice however only the ODE engine was implemented in a state where it was usable. For use in this dissertation a Bullet implementation of the interface was implemented along with the RWPE engine written from scratch, which will be the topic of chapter 3.

The *COLLADA Physics* format [40] is an open interchange format supported directly by some engines, or indirectly via the Physics Abstraction Layer. Bodies and constraints can be specified using an XML format, and the great benefit of the format is that the physics engine can be easily switched to another engine supporting the format. This is a different approach to standardisation of physics engine interfaces.

In the end it is difficult to standardise the interface for physics engines used in research. There will typically be very specific needs that must be addressed, and therefore many simulators implement their own abstraction layer to fit the specific purpose of the simulator. Efforts in standardisation is however very important and hopefully the requirements in research will find their way to implementation in standard interfaces over time.

2.2 Motion of Rigid Multi-body Systems

The theory presented in this section will primarily be based on [41, 42, 43]. The dynamics can be formulated in different ways. In the following we will consider rigid body motion on four different levels with increasing complexity. First free motion of unconstrained bodies will be considered, then holonomic constraints will be added to the formulation, and finally a contact condition avoiding penetrations will be introduced followed by contacts with friction. The latter is known as a major difficulty in simulation of rigid multi-body systems.

2.2.1 Unconstrained Motion of a Rigid Body

The motion of a body, i, is governed by the Newton-Euler equations:

$$\mathbf{M}^i(\mathbf{q}^i)\dot{\mathbf{v}}^i = \mathbf{f}^i(\mathbf{q},\mathbf{v}) + \mathbf{f}_g^i(\mathbf{q}^i,\mathbf{v}^i) \tag{2.1}$$

$$\dot{\mathbf{q}}^i = \mathbf{G}(\mathbf{q}^i)\mathbf{v}^i \tag{2.2}$$

All entities are specified in a global reference frame, and:

$\mathbf{q}^i$ is the position vector. The length varies depending on the choice of rotation representation. No assumptions are made at this point about the representation. The vector $\mathbf{q}$ is used as a notation for the position of all bodies in the system, constructed from $\mathbf{q}^1, \cdots, \mathbf{q}^n$ where n is the total number of bodies in the system.

$\mathbf{v}^i$ is the velocity vector of the centre of mass in $\Re^6$. $\dot{\mathbf{v}}^i$ is the corresponding acceleration vector. Again the notation $\mathbf{v} \in \Re^{6n}$ and $\dot{\mathbf{v}} \in \Re^{6n}$ is used for the vector of all velocities and accelerations in the system.

$\mathbf{G}$ is a mapping matrix between the velocity vector and the change in the position vector, depending on the choice of representation.

$\mathbf{M}^i$ is the mass matrix, which for a rigid body is given by $\mathbf{M}^i = \begin{bmatrix} m^i\mathbf{J}_{3\times3} & \mathbf{0} \\ \mathbf{0} & \mathbf{I}^i(\mathbf{q}^i) \end{bmatrix}$. Here the mass of the body is m^i, and the inertia given in the global reference frame around the centre of mass is $\mathbf{I}^i$. Note that the inertia is constant in the body-local reference frame, but changes in the global frame depending on the position, $\mathbf{q}^i$.

$\mathbf{f}^i$ is the net force and torque acting on the body. For an unconstrained object this will typically be a constant gravity force acting on the object. In general the force will however be considered dependent on positions and velocities of all bodies in the system.

$\mathbf{f}_g^i$ is the gyroscopic term adding a precession torque for bodies rotating around and axis not coincident with the principal inertia axes, $\mathbf{f}_g^i = \begin{bmatrix} \mathbf{0} \\ -\mathbf{w}^i \times \mathbf{I}^i(\mathbf{q}^i) \end{bmatrix}$

The Newton-Euler equation 2.2 is a second-order non-linear ordinary differential equation. Note that the gyroscopic term is a second-order term in the angular velocity and dependent on the position in a non-trivial way. For this reason some simulators choose to simply remove this term, which is valid if it can be assumed that angular velocities will be small. For a system without any contacts, constraints, springs or other interaction forces, the system is non-coupled and can be solved independently for each body. Many methods exist for solution of such problems. Instead of considering these, the interaction between multiple bodies will be considered. In this case the force, $\mathbf{f}(\mathbf{q}, \mathbf{v})$, will be coupled to some or all bodies in the system. Hence we have a *Multi-Body System* which poses new challenges.

2.2.2 Constrained Motion of a Rigid Body

If the bodies in the system must move in a certain way relative to each other, for instance for a robot with links connected by joints, the formulation can be extended with algebraic constraints. In this case we can pose a constraint on the position of bodies. This is also termed a holonomic, equality or bilateral constraint:

$$\mathbf{\Phi}(\mathbf{q}, t) = \mathbf{0} \tag{2.3}$$

In general a constraint that is defined in terms of velocity is called a non-holonomic constraint, $\mathbf{\Phi}(\mathbf{q}, \mathbf{v}, t) = \mathbf{0}$.

There are three classical paradigms to model and handle the interaction between bodies, namely the penalty-, impulse- and constraint-based simulation paradigms. The penalty-based simulator solves the Newton-Euler ODE directly assuming that the net force is constant. At each time-step the violation of the constraints $\mathbf{\Phi}$ is determined, causing a penalty force to be added in the next time-step. The force is typically given by Hook's spring law, but can also be a combination of $\mathbf{\Phi}$, $\dot{\mathbf{\Phi}}$ and $\ddot{\mathbf{\Phi}}$. Note that $\mathbf{\Phi}$ gives the amount of constraint violation, while $\dot{\mathbf{\Phi}}$ gives the velocity of change of the violation, and $\ddot{\mathbf{\Phi}}$ gives the acceleration. The main drawback of this paradigm is that avoidance of penetrations requires stiff springs, which result in large interaction forces, which in turn requires small time-steps in the integration scheme. The impulse- and constraint-based methods are conceptually very similar for systems with only holonomic constraints. Impulse-based simulation will detect a violation of $\dot{\mathbf{\Phi}}$ and apply an impulse to instantaneously correct the velocity error. A sequential approach will iterate over all constraints and correct each in turn until all are satisfied. A simultaneous approach will set up a larger equation system and solve for all impulses at the same time. The constraint-based simulation paradigm will set up an equation system that solves the Newton-Euler equations 2.2 simultaneous with a constraint like 2.3. Note that for a velocity-level formulation, the velocity cannot be corrected instantaneously, but only over time. In a system with only holonomic constraints an impulse-based simulation with a simultaneous solver and a constraint-based simulation will have very similar results. This is due to the fact that the system is smooth and the impulse and force can be easily interchanged knowing the

size of the time-step. The distinction between these two paradigms becomes clearer when considering contacts.

The ordinary differential equation in equation 2.2 together with the position-based algebraic constraint in equation 2.3 forms what is known as a Differential Algebraic Equation (DAE) with differentiation index 3 [44]. In [45] numerical solution of DAE problems is treated. Some DAEs can be solved directly using certain integration schemes, but in general DAEs with a high index is difficult to solve. One way to solve the problem numerically is to perform index reduction by differentiation of the constraints Φ. If the index can be reduced to zero, the system can be solved as an ODE. If the constraints are differentiated once, the following velocity-based constraint is obtained:

$$\frac{\delta \mathbf{\Phi}(\mathbf{q}, t)}{\delta t} + \mathbf{J}_{\Phi}\mathbf{v} = \mathbf{0} \tag{2.4}$$

Please see [41] for further details on Jacobians for different joint types. Note that this reduces the DAE index to 2, making the problem easier to solve. A solution to the original problem in 2.3 will also be a solution to the problem in 2.4. Due to errors in numerical integration however, this will not be true in practice. This has the consequence that constraints can drift over time, and special correction methods must be used to fix this drift in the positional constraints. Finally it is possible to differentiate the velocity-based constraint, 2.4 once more to get an acceleration-based constraint. In this case there can be a drift in both position and velocity, and it might not be possible to correct both at the same time.

Analytical Mechanics is an alternative approach for solving for the motion of constrained multi-body systems. Instead of considering accelerations and forces like in Newtonian mechanics, quantities like energy are considered. In Lagrangian mechanics the system is described in terms of the kinetic and potential energy. Lagrangian mechanics is very interesting in simulation of articulated structures, as it works on a minimal set of coordinates corresponding to the number of degrees of the system. The main difficulties arise in setting up the equation system, where the Newtonian approach is allows a more generic and user-friendly setup. Software does however exist for this purpose, such as [46]. Where Lagrangian mechanics is useful for articulated dynamics, it is however not very useful for modelling of non-conservative forces like friction. It works on conservation of energy, and in systems where contacts and friction should be modelled, the classical Newtonian approach is more suitable. Furthermore Lagrangian mechanics is only useful for holonomic constraints. For this reason our focus is on the Newtonian approach in this dissertation.

2.2.3 Contacts and Non-penetration Constraints

There are two aspects to contacts. First bodies can collide, causing a new contact that should be handled by special laws modelling collisions. Second contacts can be resting in which case contact forces should be applied to keep the bodies from penetrating. This often motivates the

use of hybrid simulator paradigms, which combine the impulse-based and constraint-based paradigms.

A constraint-based formulation of the dynamics for non-penetrating contacts was first done by David Baraff [47]. The formulation of the non-penetration constraint was done as a Quadratic Programming (QP) problem, where the problem is to choose the contact normal forces $\mathbf{f}_n$, such that:

$$\mathbf{A}\mathbf{f}_n \geq \mathbf{b}, \qquad \mathbf{f}_n \geq \mathbf{0} \qquad \text{and} \qquad \mathbf{f}_n^T(\mathbf{A}\mathbf{f}_n - \mathbf{b}) = \mathbf{0} \tag{2.5}$$

where $\mathbf{A}$ is a map from the contact forces to the relative acceleration in contacts, and $\mathbf{b}$ is a term of the acceleration independent of $\mathbf{f}_n$. This condition is often referred to as a Signorini condition. The conditions enforce a non-penetrating acceleration, and contact forces that only act to push bodies apart and never to pull them together. Furthermore the last condition ensures that if the bodies are accelerating away from each other, the contact force must be zero. If on the other hand the contacts are not accelerating away from each other there can be a non-zero contact force to prevent them from accelerating into each other. The holonomic constraints can be included in the problem as equality constraints. The QP problem is NP-hard in general, and Baraff suggested a heuristic method to determine which contacts where active to achieve faster solutions. So far friction has not been included, meaning that the matrix $\mathbf{A}$ is Positive Symmetric Definite (PSD).

Baraff argues that though the analytical formulation is more complex, it is also more correct and requires less iterations than penalty methods. It was suggested that the simulation is stopped at collision times, and that the time of collision is a root-finding problem. Collisions are then handled using an impulse-based method, and simulation is then restarted with new initial conditions. Instead of a simple restitution law, $v_i^+ = -\xi_i v_i^-$. Baraff suggests a method for handling simultaneous collisions by the relation $v_i^+ \geq -\xi_i v_i^-$. This gives a system of the same form as when solving for the contact forces.

The problem 2.5 together with 2.2 also forms what is known as a Differential Complementarity Problem (DCP) [48]. Typically the DCP must be linearised in a way that it becomes a Linear Complementarity Problem (LCP). In [49] the LCP is solved using Lemke's algorithm. The method has exponential complexity in worst-case, but expected polynomial running time. If the problem is inconsistent, Lemke will result in an unbounded ray, which can be used as the equivalent of an impulse. Unfortunately an unbounded ray is not a proof that no valid contact force solution exist. In frictionless systems there are however always a force solution.

Impulse-based methods rely on the application of impulses to prevent penetrations as originally proposed by Hahn [50] and later works by Mirtich [51]. The benefit of impulse-based methods is obviously that it is possible to model collisions as a discontinuous change in velocity. For small time-steps this would require infinite forces in the constraint-based paradigm. Resting contacts does however require impulses to be applied in each time-step.

For complex configurations this can be time-consuming as it can require propagation of many impulses. Implementation of a constraint-based solver and LCP solvers are more complex, but these have an advantage when it comes to larger and more static configurations. Stewart and Trinkle [52] suggested that a resting contact is modelled as an impulse train, requiring a LCP solver for the impulses. This is very similar to velocity-level constraint-based methods.

2.2.4 Friction in a Constraint-based Formulation

For realistic simulations it is vital that it is possible to model contacts with friction. When an object is at rest static friction will oppose applied forces in the contact, causing the contact to remain stationary. If the tangential forces becomes too large the object will start sliding, in which case there will be dynamic friction opposite to the direction of motion. Baraff [49] first suggested two ways of modelling contact with friction with his constraint-based formulation:

1. Using dynamic friction to approximate static friction. This involves some crawling behaviour, as a small amount of tangential velocity is needed to cause the static friction.
2. Model static friction independently for the two tangential directions. This allows friction to be exceeded in the given friction by a factor $\frac{\sqrt{2}}{2}$. It is however possible to iterate to align the tangential directions.

In [53] real static friction is considered and a modified Dantzig algorithm is presented for solving the LCP problem with static friction. Stewart and Trinkle [52] suggested using a polyhedral friction polygon for modelling Coulomb friction. This polyhedral cone approximation makes it possible to formulate the Nonlinear Complementarity Problem as a LCP. The Coulomb friction can be expressed as the following complementarity problem [41]:

$$\mathbf{d} \cdot \mathbf{f}_t^i \leq \mu f_n^i \qquad \perp \qquad \lambda_i \geq 0 \tag{2.6}$$

$$\lambda_i \mathbf{d} + \Delta \dot{\mathbf{r}}_\mathbf{t}^\mathbf{i} \geq \mathbf{0} \qquad \perp \qquad \mathbf{f}_t^i \geq \mathbf{0} \tag{2.7}$$

where $\mathbf{r}^i$ is the position of the contact, and $\Delta \dot{\mathbf{r}}_t^i$ is the relative contact velocity in the tangent direction. λ_i is Lagrange multiplier related to the relative velocity. The forces $\mathbf{f}_t^i$ and f_n^i are the tangent and normal force respectively, and μ is the Coulomb friction coefficient. The unit vector $\mathbf{d}$ expresses the direction of the tangential friction. If the contact is separating, then $f_n^i = 0$. In this case equation 2.6 shows that we must also have $\mathbf{f}_t^i = 0$. Now λ_i can take on any value, hence the relative motion can be anything. If the contact is sliding, then $\lambda_i \mathbf{d}$ must be $-\Delta \dot{\mathbf{r}}_\mathbf{t}^\mathbf{i}$ for there to be a non-zero tangent force. As we must have $\lambda_i > 0$ then the friction must also be $\mathbf{d} \cdot \mathbf{f}_t^i = \mu f_n^i$. For static friction the relative velocity is zero, and λ_i must be zero or positive. If zero, there can be a positive tangent force according to 2.7. Then equation 2.6 allows the tangential friction to stay inside the friction cone. If $\lambda_i > 0$, there can be no tangential force. In this case the complementarity is fulfilled only if $\mu = 0$. In [41] the complementarity is presented as a linear complementarity, using a linearised friction cone. The problem is that the direction $\mathbf{d}$ is unknown and must also be determined.

Anitescu and Potra [54] formulated the contact problem with friction as a LCP problem similar to Baraff. Instead of formulating the constraint on acceleration-level, they instead formulated the constraints on velocity-level. By incorporating the explicit Euler integration scheme, they formulate the problem in terms of velocities and impulses. In Baraff indeterminate configurations due to friction caused a contact impulse. Here one instead solves for the impulses directly, and the method is guaranteed to have a solution. The polyhedral friction cone is also used in this case. Anitescu and Potra [55] also considered the same problem using the implicit Euler scheme, while still being able to guarantee a solution.

2.2.5 Error Correction and Stabilisation

Integration might cause the constraints to be violated due to numerical errors in the integration scheme. If this error is not handled, the constraints will over time drift apart. Two main approaches to this problem are typically used, namely Baumgarte stabilisation and error correction by projection. Note that by using the generalised coordinate formulations it is possible to avoid handling constraints drifting apart. In the full coordinate formulation this must however be addressed. A good description of different techniques is given in [56]. Consider the acceleration-level constraints, $\ddot{\mathbf{\Phi}} = \mathbf{0}$. Instead of solving these equations, Baumgarte stabilisation solves:

$$\ddot{\mathbf{\Phi}} + 2\gamma\dot{\mathbf{\Phi}} + \gamma\mathbf{\Phi} = \mathbf{0} \tag{2.8}$$

Hence if position- or velocity-constraints are violated a small correction is added to the target acceleration. This is a very simple approach and is often used. The difficulty stems from choosing the parameters. The coordinate projection methods integrate the system and then project the errors onto the constraints manifold. This can both be done for positions and/or velocities. Correcting errors this way is more expensive, but the difficult parameter tuning of Baumgarte can be avoided.

2.3 Rigid Multi-Body Physics Engines in 3D

A number of engines have already been mentioned. For reference an overview of different open-source 3D physics engines for rigid body dynamics is shown in table 2.1. In the following

Bullet [57]	MBDyn [58]	RBDL [59]
DART [60]	Moby [61]	SD/FAST [46]
DynaMechs [62]	MuJoCo [63]	Siconos [64]
dvc3D [48]	Newton Game Dynamics [65]	Simbody [66]
IBDS [67, 43]	ODE [68]	SOFA [69]
JigLib [70]	Solfec	[71] Open Tissue [72]
MBSim [73]	PositionBasedDynamics [74]	Tokamak

Table 2.3: Overview of available open-source engines.

the engines will be introduced one by one. Some will be very briefly introduced, while others

will be more thoroughly introduced. This is in general based on the documentation provided by the projects, and investigation of the code in a few cases.

Bullet Physics is one of the most commonly used engines. Features include simulation of soft bodies, rigid bodies, clothes and similar. It uses a maximal coordinate representation, and semi-implicit Euler integration. Prior to version 2.83 gyroscopic forces were not included in the motion integration by default. Now it is by default handled using an implicit method. A sequential impulse solver is used, which in practice is mathematically equivalent to projected Gauss-Seidel method [75]. The benefit of the sequential impulse solver is that it is implemented in a decentralised manner, where building a large system matrix can be avoided. Velocity-level constraints are used and the work relies on Anitescu & Potras works [54]. By default Bullet sets a friction value per body, and the contact friction is then generated by multiplying the individual friction values. Bullet gives good customisation interfaces, so it is possible to override this behaviour for better control of friction parameters. Friction is modelled for each tangent direction independently, and is independent of normal force. The maximum friction force in this case is the friction coefficient multiplied by the latest normal force found in the iterative algorithm.

Dynamic Animation and Robotics Toolkit (DART) is a toolkit for kinematic and dynamic simulation, providing tools for control and motion planning. It is integrated with Gazebo and supports many standard scenes and geometry formats. It uses generalised coordinates for articulated multi-body systems, and uses Featherstone's Articulated Body Algorithm. Constraints are velocity-level, and the LCP formulation is used with an approximated Coulomb friction cone, and implicit time-stepping is used.

dvc3D is a maximal coordinate engine using a velocity-based LCP formulation with approximated Coulomb friction. Semi-implicit Euler is used for integration of motion. It has been implemented with a similar interface as Bullet, supporting the same collision shapes and joint types. The Bullet interface is described as well-designed, easy to use and flexible. Both a direct and an iterative solver are provided. It is the first implementation that implements the Stewart-Trinkle method exactly as described in [52]. The motivation is that the method has the desirable characteristic that it converges to the DCP as the time-step goes to zero [76]. It is suggested that the dynamics is solved using a convex QP by using an iterative algorithm that solves for the normal force and friction force in shifts.

DynaMechs supports articulated structures and higher-order integration schemes, but it does only seem to support contacts between a dynamic body and the static environment. There is no support for contacts between two dynamic bodies. Furthermore the manual states that contacts require the use of a first order integration like Euler [77]. DynaMechs does not appear to be a project in active development.

Impulse-based Dynamic Simulation (IBDS) is an engine that handles collisions and persistent frictional contacts with an impulse-based method [43]. Higher-order integration schemes, like Runge-Kutta 4, are available.

JigLib is based on an impulse-based method [78]. It uses fixed time-stepping and an semi-implicit Euler integrator. Friction is modelled with a dynamic and static friction coefficient. First the normal impulse that prevents inter-penetration is found and then the tangential impulse is found that causes the relative velocity to become zero. If this impulse is greater than allowed by the static friction model, the impulse is changed to be the size of the dynamic friction. Stacking is in general a difficult problem in impulse-based methods, and a JigLib uses a special approach for this.

MBDyn is a multidisciplinary engine using the maximal coordinate approach for multi-body dynamics. High-order integration methods are used, but unfortunately contacts are not handled by the engine. This engine is useful in mechanics, but not for simulation of manipulation actions.

MBSim is another multidisciplinary engine. It allows both event-driven and time-stepping integration and uses semi-implicit Euler for integration of motion.

Moby Has two QP-based convex solvers and a LCP solver. Allows many higher-order integration schemes, and supports generalised coordinates for articulated bodies.

Multi-Joint dynamics with Contact (MuJoCo) is an engine tailored to model-based control, biomechanics and robotics. It has been developed in parallel to the engine presented in this dissertation. It defines the dynamics in generalised coordinates, and is suitable for articulated structures. An approach is used where equality constraints are first solved for. The remaining dynamics is then projected to a tangent subspace. Projection onto the contact coordinates then allows solving for the contacts. Integration schemes supported includes semi-implicit Euler but also a modified Runge-Kutta 4 method. A different approach than Coulomb approximation is used as it does not scale well to larger problems. This also allows the formulation of a convex optimisation problem. The non-penetration constraint is solved by a cost to the optimisation function, while the friction cone is a hard constraint. MuJoCo tries to bridge the gap between the articulated dynamics that is beneficial for robot arms, and handling contacts with the environment where a full Cartesian formulation is often beneficial. The MuJoCo engine was evaluated against SD/FAST in [63], showing comparable performance in smooth dynamics scenarios.

Newton Dynamics is used for games and real-time simulation. Neither an LCP solver nor an iterative solver is used. Instead a deterministic solver is provided, which is claimed to be equally stable and fast. Investigation of the API shows that is possible to specify a static and dynamic friction coefficient. It appears that a so-called Symmetric Biconjugate Gradient solver is used.

Open Dynamics Engine (ODE) is a constraint-based maximal coordinate engine, using velocity-level constraints based on works by Stewart and Trinkle [52] and Anitescu and Potra [54]. Stewart and Trinkle suggested a LCP formulation including Coulomb friction by doing linearisation of the friction cone. This formulation did however use a position-based

constraint formulation. Anitescu and Potra did a similar linearisation of the friction cone, but formulated it as velocity-based constraints, as this will guarantee solutions. In ODE the velocity-based LCP formulation by Anitescu and Potra is used, but with a slightly different friction definition. Russel Smith argues that non-symmetric and indefinite matrices caused by friction are problematic when it comes to speed, where symmetric factorisation is twice as fast. Hence ODE does not use the linearised friction cone. Instead one of two friction models can be used. First one uses a fixed specification of the maximum friction force in two tangential directions, and is referred to as box-friction. This is independent of the normal force, and not very realistic. Second, the friction can be defined as a friction cone that is aligned with the two tangential directions. ODE first computes normal forces assuming contacts are frictionless and then the maximum friction force is determined based on the friction coefficient and the found normal force. Then ODE solves with this maximum friction force as box-friction. This means that the effective μ can deviate quite a lot from the one specified by the user, on the other hand it makes the solver robust and fast.

Instead of using a separate collision solver, Baraffs suggestion for collision handling is used [47]. Hence restitution, ξ, in ODE should be seen as part of the relation, $v_i^+ \geq -\xi_i v_i^-$. ODE will ensure a minimum outgoing velocity based on the ingoing velocity and the restitution, but the velocity is allowed to be greater than this. This has the consequence that collisions can be handled simultaneously while solving the system LCP, and that holonomic constraints and collisions can be handled simultaneously. The Dantzig LCP solver is used, based on the description in [53], where the Dantzig method was modified to cope with friction. This allows impulses to be applied rather than forces if no solution exists.

Error correction is controlled with the *Error Reduction Parameter (ERP)* and *Constraint Force Mixing (CFM)* parameters. These are basically related to the Baumgarte stabilisation parameters. The semi-implicit Euler scheme is used. If CFM is set to zero the contact or constraint will be hard, while using a non-zero CFM will make the contact or constraint soft. Soft constraints are integrated implicitly due to the Baumgarte nature.

OpenTissue is a toolkit for physics-based animation. It supports soft and rigid bodies and fluids. It targets medical applications, animation and interactive applications. The constraint-based dynamics, shock propagation and complementarity problem solvers are based on the works by Kenny Erleben [41].

PositionBasedDynamics uses a different approach to dynamic simulation than most other engines. Instead of working with constraints on the velocity level, it works directly on the position level. It is from the same author as *IBDS*, and its primary uses are described as graphics, virtual reality and games. The advantage of the method is mainly the coupling of rigid and flexibly body simulation. The method builds on projection of constraint and contact violations in positions, while keeping in mind that the resulting linear and angular momentum is conserved. Position-based dynamics is similar to the error correction by projection methods.

Rigid Body Dynamics Library (RBDL) uses the generalised coordinates approach, sup-

porting articulated dynamics. It works directly on the acceleration-level. It handles contacts but this appears only to be as equality constraints without any concept of friction. It seems to be developed mainly with biomechanics in mind.

SD/FAST is a tool for analysis and design of mechanical systems. From a specification of the system it generates the required equations in generalised coordinates. It can handle rigid bodies connected by joints, but not contacts or friction. Redundancy is handled by deleting constraints until the system has full rank. The equations of motion and constraints are formulated as a DAE of index 2, avoiding drift in velocities. To avoid drift in position, Baumgarte stabilisation is used.

Simulation and Control of Non-smooth Systems (Siconos) is a platform for simulation of non-smooth dynamical systems, including mechanical systems with contacts and friction. It provides different solvers for LCP problems, namely Lemke, PGS, QP and Conjugate Projected Gradient (CPG) methods. The main target is to provide low-level numerical routines that solves a general class of mathematical problems.

Simbody was developed for biomedical research, to overcome limitations in engines for mechanical engineering and game engines. Not surprisingly it uses generalised coordinate representations. Time-stepping involves handling of events, and constraints are handled at acceleration-level. Constraint drift are handled by projection, and many higher-order integrators are provided. Simbody suggest a different approach than SD/FAST if the system does not have full rank. Instead of just removing constraints, a least-squares solution is used if underdetermined to obtain a distributed load. For this a pseudo-inverse is calculated using a factorisation method which is similar to SVD. Contacts can be modelled with Stribeck friction, and with models of deformation.

Simulation Open Framework Architecture (SOFA) is developed for interactive simulation in a medical context. It uses a predictor-corrector approach for integration, and provides many higher-order explicit and implicit integrators. This is possible due to contacts being modelled as soft during integration. Constraint and contact forces are solved using the traditional velocity-based MLCP formulation, and used in the correction phase. The engine allows integration of finite element methods and different models for internal body forces, while maintaining ability to model for instance Coulomb friction. This linearisation is in fact very closely related to the *CFM* and *ERP* parameters used in ODE, and it makes sense to use such parameters when soft bodies are being simulated. After integration of the free motion, a correction is performed in both velocity and positional space by solving a MLCP problem, giving the Lagrange multipliers that correct all constraints and contacts.

Solfec uses velocity-level time-stepping and a Gauss-Seidel constraint solver. A block solver is used to make parallelisation possible. The engine has been used for modelling dynamics of large, densely packed assemblies with stiff non-convex bodies. The integration

scheme works as follows:

$$\begin{aligned} \mathbf{q}_{n+\frac{1}{2}} &= \mathbf{q}_n + \frac{h}{2}\dot{\mathbf{q}}_{n+1} \\ \dot{\mathbf{q}}_{n+1} &= \dot{\mathbf{q}}_n + h\mathbf{M}^{-1}\mathbf{f}^{ext}_{n+\frac{1}{2}} + \mathbf{M}^{-1}\mathbf{H}^T\mathbf{f}_{n+\frac{1}{2}} \\ \mathbf{q}_{n+1} &= \mathbf{q}_{n+\frac{1}{2}} + \frac{h}{2}\dot{\mathbf{q}}_{n+1} \end{aligned} \tag{2.9}$$

First positions are integrated half a step, and then new constraints are updated and new contacts are found. The constraints are solved for and the velocity after an entire step is found. Finally the position is updated using the updated velocities. The Signorini-Coulomb law is used, indicating that Coulomb friction is supported.

Tokamak is an impulse-based engine for real-time use in games. It uses semi-implicit Euler integration, and its API seems to suggest support for Coulomb friction.

In manipulation it is important to be able to handle resting contacts efficiently. Contacts should be modelled with both restitution and friction, motivating a hybrid approach between impulse- and constraint-based simulation paradigms. Also we will prefer a method that works in force-space, as control strategies and simulated tactile sensors will work on forces. To summarise our findings in existing engines, we will briefly draw the attention to the ideas we share, and on which RWPE are built. MuJoCo avoids the Coulomb approximation similar to what we do in RWPE. Also the problem is posed as a convex optimisation problem making solution of the problem efficient. The Simbody approach uses a pseudo-inverse to obtain a least-squares solution distributing loads in case of redundancy. Though Sofa has a focus on soft bodies, it has a wide range of integration schemes available, and uses a predictor-corrector approach to enforce constraints while allowing higher-order integration. The Solfec engine focuses on assembly and proposes a higher-order method for this purpose, which is also related to the integration scheme used in RWPE. In chapter 3 it will be discussed in detail how RWPE combines some of the mentioned views.

2.4 Previous Physics Engine Comparisons

Previous engine comparisons have been performed which will be considered here. It should be noted that some of the previous comparisons are some years old. Many engines have evolved a great deal since then and are still under active development. Hence some of the previous evaluations might no longer hold true, which will also become apparent in chapter 5 where some of these tests have been run again for ODE, Bullet and RWPE. Keeping this in mind, we will now look into some of the previous findings.

An evaluation of open-source engines was done in [79] with focus on features, scalability and stability. Ten engines were rated based on features, documentation and usability. Seven of these are now open-source. From the criteria set up, Newton, ODE, and Novodex were selected for comparison. All three proved to handle static friction well, except that they

all used Coulomb friction pyramids. A test of integration of gyroscopic force revealed that ODE gains angular momentum causing it to become unstable. When it comes to restitution models, it was proven that Newton uses damping causing it to constantly lose energy. ODE on the other hand gave fairly good results. On the other hand Newton seems to be better at simulation of a pendulum than ODE. A serial chain with a number of objects was constructed to test constraint stability. Here it was found that ODE could handle more objects in series than Newton before going unstable. Overall Newton and ODE was found to be equally good in an overall qualitative evaluation, the Novodex did however chosen as the one giving the best results.

In [80] seven engines were evaluated in the context of the PAL framework with focus on simulation for games. Of these we will focus on Bullet, JigLib, Newton, ODE and Tokamak which are open-source. A linear free-falling test proved Newton to have a large deviation from the expected trajectory due to internal damping. A restitution test proved Bullet bouncing too much, and the four others bouncing too little for different restitution values. Newton handled static friction best, while Tokamak performed worst. It was found that constraints were best simulated by ODE, while difficult for JigLib. On the other hand ODE used more time. Some of these results are not too surprising considering the difference between impulse- and constraint-based simulation. In chapter 5 some of these tests will be run again for ODE, Bullet and RWPE, showing different results. Hence the evaluation is probably outdated for the engines that are still under active development to this day.

In [81] the engines Bullet, Havok, MuJoCo, ODE and PhysX were compared. The focus in this comparison is more challenging stress tests than the tests performed in [79, 80]. The engines were tested in four scenarios, with the most interesting for us being a grasping scenario where a capsule is grasped with a dexterous gripper followed by a motion of a robot arm. In this case MuJoCo and ODE was proven to be the fastest, while Havok, Bullet and PhysX used roughly twice the time. Note that the iterative solver of ODE was used for these tests, and not the direct LCP solver as used in this dissertation. The simulations were also performed with different sizes of the time-step to check consistency. It was found that the MuJoCo engine performs orders of magnitude better than the other engines, and that both ODE and Bullet has problems for large time-steps where no results could be generated due to instability. One interesting finding is that while the kinetic energy and angular momentum is conserved quite well in the MuJoCo engine, the linear momentum is not. This is due to the fact that MuJoCo works in joint space and not in Cartesian space. Finally it was found that even though Bullet provides articulated dynamics, this seems to not yet work stable enough with contacts.

2.5 Modelling of Rigid Bodies & Materials

Modelling the geometries and dynamic properties of rigid bodies is important for realistic simulation. Especially in tight-fitting scenarios it will be expected that this has a large influence on the outcome of simulations. In this section some of the traditional methods will be considered.

2.5.1 Geometry

Typically objects are modelled either as a boundary representation (B-rep) or as a solid. The most commonly used type of B-rep is the triangle mesh, where a solid object is represented as an outer shell. The objects in figure 1.1a are illustrated in 2.1 where the triangle mesh representation is clear. In this example the cylinder is represented by 80 triangles and the

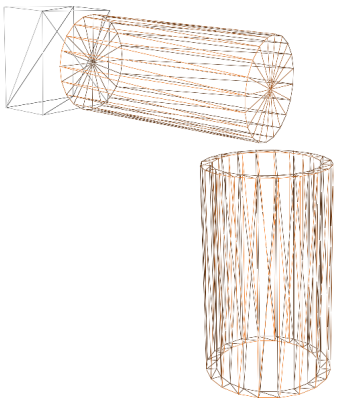


Figure 2.1: Example of objects in a triangle mesh boundary representation.

tube is represented by 160 triangles. In general a boundary representation is composed of two components, namely the geometry and the topology. Geometry is given as points, lines, curves and surfaces such as triangles, other polytopes, or curved surfaces like NURBS. The topology on the other hand is a graph of vertices, edges and faces. In the general case geometry is often given without any topological information in which case it is often referred to as a polygon soup. Strictly speaking a polygon mesh should have topological information, such that for instance shared vertices or neighbouring faces can be determined.

A solid represents an object using primitives. These are typically geometries available when modelling object in Computer Aided Design (CAD). Constructive Solid Geometry (CSG) allows combining these primitives using different set operations like union, intersection, and difference. Objects modelled this way are always water-tight, unlike B-rep where the representation can have holes in it. Especially B-rep representations reconstructed based on stereo imaging, lasers and point clouds or similar can be incomplete.

Each of these representations has their own benefits and drawbacks. Often a solid representation exists only during the CAD phase. If the objects are then to be used in simulation they are often triangulated and exported as a triangle mesh as this is the most commonly

supported format. The same goes for more detailed B-reps like NURBS, which will also typically be converted to the triangle mesh. Unfortunately the triangle mesh representations will often convert otherwise smooth surfaces to many points, lines and planar surfaces. This causes non-differentiable surfaces, meaning normal determination becomes difficult. This is a major drawback for rigid body simulators that relies on the normal information. In [49] this problem in contact handling was posed, and 25 years later this remains an issue.

Many formats exist for interchange of geometry and associated graphical information. In RobWork, the Open Asset Import Library [82] is used. This library supports different geometry formats. Also on-going work involves integration of NURBS interchange by using the openNURBS SDK [83], as part of our efforts towards the use of more smooth geometric models.

2.5.2 Inertial Parameters

Besides the geometric definition of objects, the dynamical parameters must also be determined to realistically simulate the motion of the objects. From equation 2.2 we see that the mass matrix, $\mathbf{M}^i$, is important. Hence the mass, m^i, and the inertia, $\mathbf{I}^i$, must be determined. Note that the inertia should be given in the centre of mass of the body. Hence the centre of mass, $\mathbf{p}^i$ must also be determined. These three parameters will be referred to as the inertial parameters of a body. The mass will typically be the easiest parameter to estimate. The centre of gravity and inertia is however often based on estimates. For bodies modelled with solid geometry these parameters are easily determined analytically. This is however much more difficult for non-trivial triangle meshes. For this purpose Mirtich presented a method for calculation of the inertia from a triangle mesh [84]. This does however assume that the mesh is well-behaved and water-tight.

2.6 Contact Detection

The most important prerequisite for doing correct simulations is correct detection of contacts between bodies. If the detected contacts are wrong, it will be impossible to do accurate computations of the dynamics in the system. Usually detection of contacts is termed collision detection. In this thesis we will make the distinction between contact and collision detection. The latter is a method for determining if bodies are overlapping (either true or false), while contact detection is the determination of complete contact information including contact points on the two bodies, a normal and a depth or distance. Hence a contact between two bodies does not necessarily mean that there is a collision, but there might be.

Both Bullet and ODE implements its own contact strategies for pairs of certain primitives, and provides interfaces for the user to implement own contact strategies, as well as using the contact detection without using the simulator as such. In principle contact detectors and dynamics engines can then be combined in different ways. Contact detectors are however often implemented such that they work best with a certain engine. An overview of different collision detection and proximity query packages are given at [85]. To mention a few:

PQP is based on the use of a Bounding Volume Hierarchy (BVH) of Oriented Bounding Boxes [86] and Rectangle Swept Spheres [87]. The former is beneficial for overlap tests, while the latter is cheaper for distance and tolerance tests.

GJK The *Gilbert–Johnson–Keerthi* algorithm is used for distance calculation between convex sets [88]. The enhanced GJK method follows vertices on the surface of object to find the local minimum in distance. This of course requires the topology to be known.

OPCODE uses trees of Axis Aligned Bounding Boxes (AABB). This is the default in ODE.

GIMPACT is integrated with both ODE and Bullet. Has support for concave meshes and deformable bodies, and is used for detection of mesh to mesh geometries in Bullet, as well as for concave and compound shapes.

RAPID The *Robust and Accurate Polygon Interference Detection* software is a small package that works on polygon soups. It is based on OBB-Trees, and outputs triangle pairs.

V-COLLIDE is a higher-level interface that allows queries for multiple bodies. RAPID is used for generation of triangle pairs in contact.

I-COLLIDE For large environments with many convex watertight polytopes, I-COLLIDE is useful. Topological information is however required. Distances are reported, where RAPID gives only triangle contact-pairs.

In [89] a PQP-based method was used for contact generation. It was found that OPCODE and GIMPACT had problems when it comes to dynamic simulation with ODE and Bullet in locomotion and manipulation scenarios, which the PQP-based method with contact clustering solved. In RobWorkSim a similar PQP-based approach has traditionally been used when doing simulations with ODE.

One of the major difficulties in contact detection for polyhedral models, is handling of penetrations. This is due to the fact that penetration depth is very difficult to determine, in particular when objects have arbitrary non-convex shapes. This is typically solved by avoiding penetrations by using a contact layer. One important feature of polyhedral representations is that these can always be decomposed into convex subparts.

2.7 Summary

It was found that a wide range of different toolkits exist for simulation of mechanical structures and robotics in particular. Many of these are targeted classical robotics, while others are developed with other applications. Even though the focus is different, all tend to rely on some common underlying physics engines for doing dynamics, and this is almost always either ODE or Bullet. Toolkits for multi-agent systems are developed with the purpose of running many independent simulations in parallel, while researchers in biomechanics and humanoids prefer engines using certain articulated formulations of the dynamics. In games the speed is important, and for manipulation realistic interaction modelling is important.

CHAPTER 3
A New Engine for Manipulation

A new rigid body dynamics engine is presented, addressing some of the challenges in simulation of manipulation actions. The engine is composed of a large number of different algorithms and modules, but before considering each of these in detail it is necessary to give a broad overview of how the overall design works. This will give an idea of how the individual modules work together and fits into the overall simulation loop. First of all, some common notation and terminology is introduced in section 3.1. The overall simulation loop is then described with a brief introduction to the individual components in section 3.2. In sections 3.3 to 3.11 the individual modules are presented in detail, and finally an in-depth discussion of the entire simulation loop is given in section 3.13. The name of our new engine is *RobWorkPhysicsEngine* (RWPE) as it is an engine integrated into the RobWork framework. In the future it is, however, our hope that it is integrated into even more frameworks than just RobWork.

3.1 The Body-Constraint Graph

The concept of a contact graph is used to decompose the dynamics into smaller independent sub-problems that can then be solved more efficiently. In [41] the concept is explained in detail. Also, the contact graph has an important role when collisions are handled with shock propagation, for instance, as in [78].

In RWPE we model and solve the dynamics a bit different than most other engines, but the same efficiency considerations apply regarding decomposition of the dynamics into smaller sub-problems. Furthermore collisions are handled sequentially when possible. This motivates the use of a similar graph, which we will call the *Body-Constraint Graph* (BCG), in RWPE. Notice that in this case the term *constraint* covers contacts and springs as well. In this section the graph concept is introduced as a means to introduce some useful notation as well as useful graph operations.

3.1.1 Notation

The Body-Constraint Graph is what can be described as a directed and labeled multigraph. This graph is given by the 8-tuple:

$$G = (\Sigma_V, \Sigma_E, \mathbb{V}, \mathbb{E}, S, T, L_V, L_E) \tag{3.1}$$

where:

Σ_V is a fixed set of node labels. The label **static** is used for bodies that remain stationary during simulation. For bodies with velocities that are controlled directly by the user, the **kinematic** label is used. The motion of dynamic bodies are not affected by other bodies, but other bodies are affected by the motion of kinematic bodies. Finally, the label **dynamic** is used for bodies with motion that is governed by dynamic motion equations. The motion of dynamic bodies can only be controlled indirectly via constraints, springs or contacts with other bodies.

Σ_E is a fixed set of edge labels. An edge must either be labelled as a **constraint**, **contact**, **new contact** or **spring**. Notice that when two bodies come into contact, a new edge is created with the label **new contact**. This indicates that a collision has occurred. Persisting contacts are instead labelled **contact**, indicating that no collision has occurred. Notice that all edges connect two nodes and that at least one of these two nodes must be labelled as **dynamic**.

$\mathbb{V}$ is the set of all bodies, $V^1 \cdots V^m$. The set is composed of the union of multiple subsets, $\mathbb{V} = \mathbb{V}_S \cup \mathbb{V}_K \cup \mathbb{V}_D$. Here the subscript refers to the related label, Σ_V, as given above.

$\mathbb{E}$ is the set of constraints and contacts, $E^1 \cdots E^n$. The set is composed of the union of multiple subsets, $\mathbb{E} = \mathbb{E}_e \cup \tilde{\mathbb{E}}_i \cup \mathbb{E}_i^{new} \cup \mathbb{E}_s$. Here $\mathbb{E}_e$ is the equality edges (constraints), $\mathbb{E}_i$ is the inequality edges (the contacts) and $\mathbb{E}_s$ is the spring edges. The notation $\mathbb{E}_i = \tilde{\mathbb{E}}_i \cup \mathbb{E}_i^{new}$ will be used for the combined set of persistent and new contacts.

S is a map that gives the source of an edge $E \to V$. The shorthand notation $S^i = S(E^i)$ is used to refer to the source of edge i.

T is a similar map that gives the target of an edge $E \to V$. The shorthand notation $T^i = T(E^i)$ is used to refer to the target of edge i.

L_V is the map from a node to its label $V \to \Sigma_V$.

L_E is the map from an edge to its label $E \to \Sigma_E$.

Notice that the direction of an edge is important mainly for contacts that have a directed normal. The notation (S^i, T^i) is used to refer to the source and target of edge i as an ordered pair of nodes. Notice that $(S^i, T^i) \neq (T^i, S^i)$. In case the direction of an edge is not important the notation $\{S^i, T^i\}$ is used for an unordered pair of nodes. In this case $\{S^i, T^i\} = \{T^i, S^i\}$.

3.1.2 Graph Dynamics

One motivation for the graph concept is to allow splitting the dynamic problem into sub-problems. This is similar to a decomposition of the graph into sub-graphs. Notice that two sub-graphs that can be solved independently, might again come into contact with each other. In this case, the sub-graphs must be merged back into one graph again. It is important to be aware of the dynamically changing graph decomposition and the frequency of change. Splitting and merging graphs to exploit better efficiency in the dynamics solver might be infeasible if it happens too often. In table 3.1 an overview of possible changes in the graph is given along with an assessment of the expected frequency and impact of such changes. The

Vertices			
Change	**Affects**	**Freq.**	**Description & Implications**
Add Body	$\mathbb{V}, L_V$	Low	User is expected to add bodies that are not in contact initially. This has low implication as the body is initially independent.
Remove Body	$\mathbb{V}$, $\mathbb{E}$, S, T, L_V, L_E	Low	Can potentially have a dramatic impact. It is however expected that removal of a body happens when it has become insignificant to the simulation.
Kinematic $\leftrightarrow$ Dynamic	L_V	Low	Has implications on the dynamics solver.
Static $\leftrightarrow$ Kinematic	L_V	Low	Has implications on the collision solver. Static bodies can be considered one body, kinematic bodies can not.

Edges			
Change	**Affects**	**Freq.**	**Description & Implications**
Add Constraint Remove Constraint	$\mathbb{E}_e$, S, T, L_E	Low	Has implications on the dynamics solver.
Add Spring Remove Spring Change Spring	$\mathbb{E}_s$, S, T, L_E	Low	Springs do not impact the dynamics and collision solver, but do require time synchronisation.
Add New Contact Remove New Contact	$\mathbb{E}_i^{new}$, S, T, L_E	High	Has implications for the collision solver.
New Contact $\rightarrow$ Contact	L_E	High	After collisions the contact becomes persistent and part of the dynamics. This makes the dynamics harder.
Remove Contact	$\tilde{\mathbb{E}}_i$, S, T, L_E	High	Leaving contacts make the dynamics simpler.
Spring $\leftrightarrow$ Constraint	L_E	Low	User-specified switch between compliance and fixation. Springs cause dynamics decoupling and does not carry impulses. Graph should be splitted or merged whenever possible.

Table 3.1: Overview of possible dynamic changes in the Body-Constraint Graph.

changes in the graph that happen most frequently are the changes in $\tilde{\mathbb{E}}_i$ and $\mathbb{E}_i^{new}$. It will become apparent in section 3.2 that these can in fact change multiple times inside a single step. Because of this, the set of contacts is stored in an external state structure, such that the Body-Constraint Graph is a constant structure during a simulation step. The state of the graph, G, is then stored as the tuple:

$$G^S = (\mathbb{E}_i, S_{\mathbb{E}_i}, T_{\mathbb{E}_i}, L_E^{\mathbb{E}_i}) \tag{3.2}$$

3.1.3 Operations on the Graph

In the following sections it is important to be able to perform certain operations on the graph. Besides the fairly straight-forward operations of adding, retrieving and removing nodes and edges in the graph, we will focus on three methods for retrieving certain significant sub-graphs.

- DynamicComponents(G,G^S) finds sub-graphs of a given graph G with graph state GG^S. The decomposition is performed in two steps. First, each edge i in $\mathbb{E}$ is visited. If the source body S^i is dynamic and the body is already present in an existing component, the edge is added to this component. If not, the same check is performed for T^i if the target body is dynamic. If neither the source nor the target body is found in an existing component, a new component is created with this single edge. The source and target can also lie in two different components in which case the two components are merged into one. In the second step, the edges in each component are traversed and if an edge connects a dynamic body to a non-dynamic body, the non-dynamic body is added to the component as well. Note that each sub-graph will now have disjoint sets of dynamic bodies, while the static or kinematic bodies can exist in multiple sub-graphs simultaneously. These sub-graphs are then the *dynamic components*. The impulses, forces and motion in each dynamic component can not influence bodies in a different component. This allows independent solution of both collisions and dynamics. The motivation for this method is discussed in section 3.1.4.

- DynamicComponentsSpringDecoupling(G,G^S) is an algorithm very similar to the one described above. Instead of traversing all edges, only the edges $\mathbb{E}_e \cup \mathbb{E}_i$ are traversed. This makes separate components that are dynamically dependent on each other, but only through a soft spring dependency. In section 3.10 the use of this decomposition is motivated in connection with the contact and constraint solver.

- ConnectedComponents(G,G^S,$\{S^1, T^1\}, \cdots, \{S^k, T^k\}$) is an algorithm that creates connected components based on a set of source-target body-pairs. For each pair, $\{S^i, T^i\}$, a new graph is created including all edges, $\mathbb{E}(S^i, T^i)$. If two graphs have a shared body, they are merged into one. This component search is used as a part of a collision propagation method described in section 3.5.2. Here the purpose is to create the smallest possible sub-graphs of the edges, $\mathbb{E}_i^{new}$. Connected components will typically be a second step after finding dynamic components with spring decoupling as described above.

3.1.4 Hierarchical Engine Design

Based on the previous discussion, a hierarchical engine design with three levels is proposed:

$$\text{Physics} \rightarrow \text{World} \rightarrow \text{Island}$$

In this dissertation, only the island level is considered. The design of the island level is, however, motivated by the fact that the island should be considered a sub-component of the world level. Similarly the physics level can be composed of multiple independent worlds. Worlds will always be completely decoupled from each other, and the purpose of the physics level is to step each world forward such that the time remain relatively synchronised during simulation. Using the *physics level* requires the user to explicitly define independent worlds. An example of this is process simulation of completely independent robot cells. Even if a simulated object needs to be passed between cells, this happens infrequently. This can therefore be controlled explicitly by the user by moving an object from one world to the other. The *world level* will automatically decompose the problem into independent islands using the DynamicComponents method. Islands are dynamic entities that must be split and merged during simulation to optimise the computational efficiency. Islands do, however, have independent time, and the difficulty by merging the islands is the time synchronisation required. Worlds are responsible for optimisation by automatic sub-division into islands as well as merging islands that come into contact with each other by time synchronisation, merge of body-constraint graphs and merge of the island state structures. For efficiency it is also important that the world limits the number of split and merge operations, as such operations can be expensive in itself.

The world and physics levels are not discussed further, but the island level is the topic of sections 3.2 to 3.13. It is important to be aware of the context of the island engine, which motivates the graph representation and the methods for obtaining certain sub-graphs.

3.2 Overall Design

The overall engine design is illustrated in figure 3.1. We will now consider each component in the same order as shown in the loop. In this section, the components are briefly introduced to make the overall purpose of each component clear. In each of the following sections we subsequently discuss the components in much greater detail. One step of the simulation loop

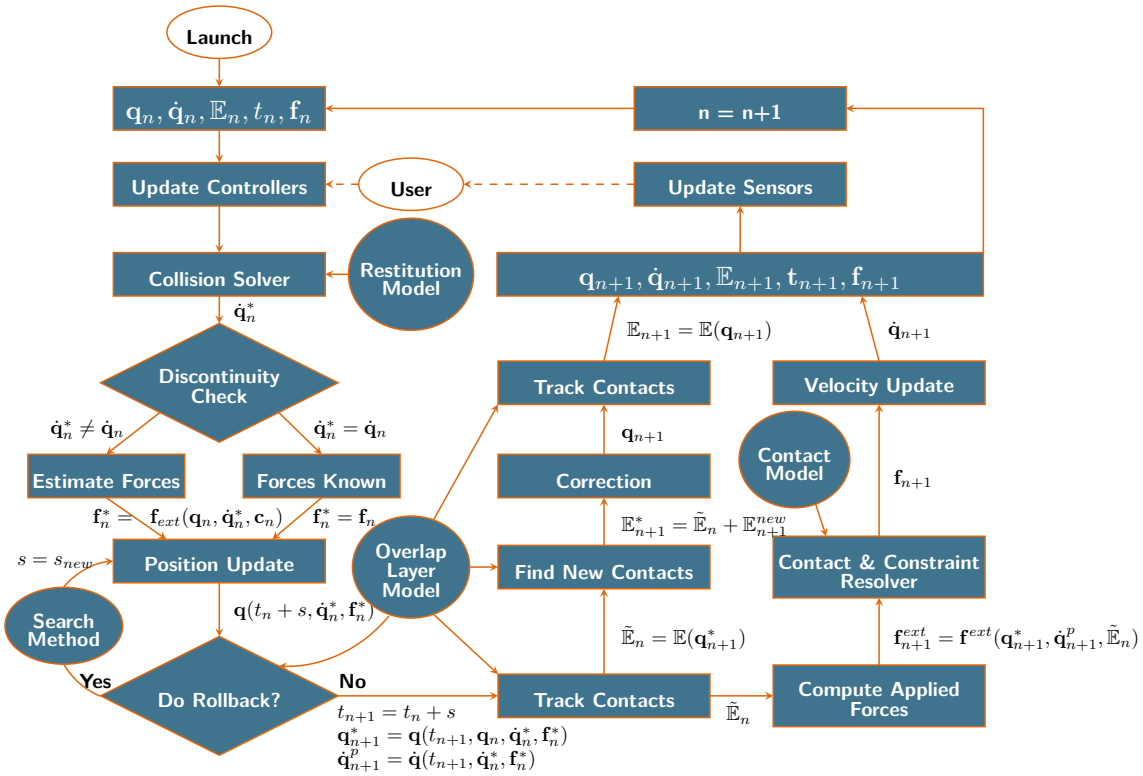


Figure 3.1: An overview of the overall simulation loop

begins at *Launch* at the top left. The initial state of the system is given as input, including the body positions, $\mathbf{q}_n$, velocities, $\dot{\mathbf{q}}_n$, contacts and constraints, $\mathbb{E}_n$, the current time, t_n and the body net forces and torques, $\mathbf{f}_n$, at time t_n. The task is now to determine the new positions, velocities and forces at time t_{n+1}. To determine these the following steps are performed:

Update Controllers (section 3.3) gives the user the ability to control the motion of bodies by different means. The controllers provide interfaces to the user on different abstraction levels. This can be controlling the motion of a robot arm or making a kinematic body move towards a target configuration. A good controller for a robot arm could even model the physical controller for a certain physical robot, respecting velocity and acceleration limits for that particular robot. The controller takes the user input and transforms it into quantities that are controllable in simulation. Controllable quantities include velocities of kinematic bodies, joint velocities and external force and torque applied to dynamic objects. This step is performed before the collision solver to allow the kinematic velocities to be changed in a discontinuous manner, causing a different collision handling if there are colliding contacts.

A *Restitution Model* (section 3.4) determines how new colliding contacts are handled. The *Collision Solver* (section 3.5) will apply impulses at contacts where bodies collide, which will cause a discontinuous change in velocities. The relative velocity in all contacts will be non-penetrating after this step. If impulses have been applied, the new velocities, $\dot{\mathbf{q}}_n^*$, will be different than the input, $\dot{\mathbf{q}}_n$. As the forces, $\mathbf{f}_n$, are only valid for $\dot{\mathbf{q}}_n$, an approximated force, $\mathbf{f}_n^*$, is constructed based on the external forces, which includes gravity, spring forces and forces applied directly by controllers. This means that all contact and constraint forces are assumed zero after impulses have been applied. If, on the other hand, there was no collisions, the forces are known and $\mathbf{f}_n^*$ is given directly as $\mathbf{f}_n$.

Position Update (section 3.6) and *Rollback* (section 3.7) are integrated in a loop that tries to integrate the position one timestep forward using the new velocities, $\dot{\mathbf{q}}_n^*$, and forces, $\mathbf{f}_n^*$. After updating the position to time t_{n+1} new contacts might occur. It is likely that such new contacts are in penetration and that they have colliding relative velocities. As penetrations should be avoided, the time-step is decreased to s when updating the position to the exact time of impact. A rollback method is used to perform a root search for the correct value for s. To integrate position, either a first order explicit Euler method or a second-order Heun's method is used. The former is only used in case the collision solver has colliding contacts. If there were no colliding contacts, $\mathbf{f}_n$ is valid and a higher-order integration can be used. The output is new positions, $\mathbf{q}_{n+1}^*$, a predicted velocity, $\dot{\mathbf{q}}_{n+1}^p$, and the new time t_{n+1}.

Updating the positions will cause contacts to move at the surface of the bodies. In section 3.9 a friction model is proposed that relies on stateful contacts. *Tracking of Contacts* is used to track the persistent contacts to their new location, $\tilde{\mathbb{E}}_n$, while maintaining this internal state. Notice that new, non-tracked contacts are added directly to $\tilde{\mathbb{E}}_n$ if their relative ingoing velocity is small enough and there is no collision. If a tracked contact has relative velocity that causes the contact to separate, it is removed from the set. Hence contact tracking is based on both contact position and velocity.

Notice that the loop splits into two separate tracks at this point. The left track will work only on the positions and contacts, while the right track will work only on the velocities and forces. Only the persistent contacts are used for determination of new forces and velocities in the right track. This is due to the fact that new colliding contacts must first be handled at time t_{n+1} by the *Collision Solver*.

In the left track, *Find New Contacts* will determine the new contacts, $\mathbb{E}_{n+1}^{new}$, which is contacts that are colliding due to their relative velocities. These new collisions must be handled by the collision solver and are first treated in the next step of the loop. *Correction* (section 3.8) is then performed on the complete set of both persistent and new contacts, $\mathbb{E}_{n+1}^*$, to correct the positions of all bodies, such that a positional drift in contacts and constraints is corrected. Correction by projection is used in this case, and the correction will cause the final position of bodies, $\mathbf{q}_{n+1}$. A new tracking is then performed. Here the complete set of contacts, $\mathbb{E}_{n+1}^*$, is tracked to $\mathbb{E}_{n+1}$ after correction. Unfortunately, this projection can cause

new contacts on its own. This has in particular been experienced in tight-fitting assembly situations. If this happens, correction is repeated, with the new contacts included, until no more new contacts are detected. If the contact set has been changed fundamentally, this will typically also require that the right track is re-evaluated. We find that such behaviour is due to inadequate contact detection in tight-fitting scenarios, and for this reason we conceptually stick to the loop as presented in figure 3.1. In section 3.13 we will get back to this issue.

For the right track, *Compute Applied Forces*, is first performed. This finds all external forces acting on bodies, $\mathbf{f}_{n+1}^{ext}$, as well as spring forces, $\mathbf{f}_{app}^{\alpha}$. A *Contact Model* (section 3.9) is used to model friction phenomena. The *Contact & Constraint Resolver* (section 3.10) will resolve all contact and constraint forces such that given motion constraints are satisfied at time t_{n+1}. Notice that only the existing contacts are handled as they are assumed to give rise to contact forces over the complete time-step. New contacts detected in the left track have occured at time t_{n+1} and will be handled by the collision solver in the next step from t_{n+1} to t_{n+2}. The result of the contact and constraint force resolution is that the net force and torque, $\mathbf{f}_{n+1}$, is determined for all dynamic bodies, $\mathbb{V}_D$. Finally, the found net forces can be used in the *Velocity Update* (section 3.6) to determine the resulting velocity, $\dot{\mathbf{q}}_{n+1}$. As before the integration scheme is either explicit Euler or Heun.

The new state at time t_{n+1} has now been determined. This is used as a input for the next step and to *Update Simulated Sensors* (section 3.11). The simulated sensors will give tactile information that the user can read and use to set new controller targets.

Note that the modules *Find New Contacts* and *Track Contacts* are treated as one in section 3.12 as the motivation will be better understood at this point. This concludes the overall walk-through of the simulation loop. In the following sections, each of the different modules will be considered individually in much greater detail.

3.3 Update Controllers

Simulated controllers can be seen as the user input to the system. These control how bodies move during simulation. In table 3.2 an overview of a few standard controllers is given.

<table>
<tr><th>Controller</th><th>Works On</th><th>User Input (Target)</th><th>Controlled Entity</th></tr>
<tr><td rowspan="2">Body</td><td>Kinematic Body</td><td>Body position, $\mathbf{q}_n^i$, trajectory or velocity, $\dot{\mathbf{q}}_n^i$</td><td>Body velocity, $\dot{\mathbf{q}}_n^i$</td></tr>
<tr><td>Dynamic Body</td><td>Body position, $\mathbf{q}_n^i$, trajectory or force, $\mathbf{f}_{ext}^i$</td><td>Body force, $\mathbf{f}_{ext}^i$</td></tr>
<tr><td>BeamJoint</td><td rowspan="7">Dynamic Device</td><td>Deflection angle</td><td rowspan="5">Joint velocity, $\Delta\mathbf{v}^{\alpha}$</td></tr>
<tr><td>PD</td><td>Joint position, $\Delta\mathbf{r}^{\alpha}$, and velocity, $\Delta\dot{\mathbf{r}}^{\alpha}$</td></tr>
<tr><td>Pose</td><td>Pose and velocity screw</td></tr>
<tr><td>SerialDevice</td><td>Cartesian position and velocity
Joint position and velocity</td></tr>
<tr><td>SyncPD</td><td rowspan="3">Joint position, $\Delta\mathbf{r}^{\alpha}$</td></tr>
<tr><td>VelRamp</td><td></td></tr>
<tr><td>SpringJoint</td><td>Joint force, $\mathbf{f}_{app}^{\alpha}$</td></tr>
</table>

Table 3.2: Overview of available controllers in RobWorkSim.

There are two major types of controllers: one type for controlling bodies and another type for controlling joints of dynamic devices. A dynamic device can simply be considered a collection of dynamic bodies connected by constraints (joints).

The *body controller* works differently depending on the type of object controlled. If the controlled body is a kinematic body, the user sets a target position, trajectory or velocity as input to the controller. The velocity target is trivial as the controller can pass it directly to the engine. To follow a trajectory, the controller will calculate the deviation from the trajectory in each step and adjust the velocity of the body such that the error is corrected during the time-step, Δt. Strictly speaking the time-step is unknown at this point as rollback has not yet been performed. As control is on the velocity level, we do not find this critical enough to justify an iterative approach for adjusting the controllers with adjusted time-steps during rollback. Notice that the user is also able to specify a position target. In this case, a trajectory is automatically generated by ramping up velocity to a maximum linear and angular velocity limit, $\|\dot{\mathbf{p}}^i\|_{max}$ and $\|\mathbf{w}^i\|_{max}$, while respecting the maximum acceleration limits, $\|\ddot{\mathbf{p}}^i\|_{max}$ and $\|\dot{\mathbf{w}}^i\|_{max}$. For dynamic bodies a position and trajectory target can be specified in a similar way. For dynamic bodies the controlled entity is an external force applied to the body. Ideally the external force is controlled such that the deviation from the desired trajectory is corrected during the next time-step. A desired velocity is determined, but in this case the velocity can not be controlled directly. Instead an external force must be applied to the body to correct the velocity during the next time-step. The external force applied to the body is proportional to the desired velocity correction. Notice that, on the contrary to the kinematic control, this type of control is much more sensitive to a change in the time-step during rollback. For this reason the use of positional control of dynamic bodies is strongly dis-encouraged.

A range of different *dynamic device controllers* is available. These controllers are not discussed in detail, but they work similarly to the already discussed body controllers. Different types of position- and velocity-level targets can be set on the controllers, which will then control the joint velocity directly or apply a joint force to achieve desired joint positions.

It is worth noting again that the output from the controllers is also the input to the engine. Hence the controller interface, provided by the engine, influences which types of controllers that can be used on top. The entities in the right column of table 3.2 are supported by our new engine, and users can implement their own controllers to simulate the specific behaviour they need. In general, we dis-encourage the use of controllers that apply forces, as this gives unrealistic control of dynamic bodies and causes problems when the time-step is not known. Hence the signal from the controller to the engine should be either the direct velocity of kinematic bodies or the target motor velocity in joints. Instead of forcecontrollers we introduce compliant joints. These are modelled with better support for the changing time-steps due to rollback, and we find such control more physically justified and flexible.

3.4 Restitution Model

If there are new colliding contacts in the contact set $\mathbb{E}_i^{new}$, the response to these collisions will be governed by a restitution model. Consider a colliding contact $E^\alpha \in \mathbb{E}_i^{new}$ between the bodies $V^s = S(E^\alpha)$ and $V^t = T(E^\alpha)$. The velocity of the contacts at the source and target bodies is given as:

$$\dot{\mathbf{r}}_{in}^{\alpha,s} = \dot{\mathbf{p}}_n^s + \mathbf{w}_n^s \times (\mathbf{r}_n^{\alpha,s} - \mathbf{p}_n^s) \qquad \text{and} \qquad \dot{\mathbf{r}}_{in}^{\alpha,t} = \dot{\mathbf{p}}_n^t + \mathbf{w}_n^t \times (\mathbf{r}_n^{\alpha,t} - \mathbf{p}_n^t) \tag{3.3}$$

where $\mathbf{p}_n^i$ is the linear part of the position vector $\mathbf{q}_n^i$. The linear and angular part of the body velocity, $\mathbf{v}_n^i$, are $\dot{\mathbf{p}}_n^i$ and $\mathbf{w}_n^i$ respectively. The position of the contact E^α at body V^i in world coordinates is $\mathbf{r}_n^{\alpha,i}$. The velocity difference is given as $\Delta\dot{\mathbf{r}}_{in}^\alpha = \dot{\mathbf{r}}_{in}^{\alpha,t} - \dot{\mathbf{r}}_{in}^{\alpha,s}$, and as the contact is colliding then $\mathbf{n}^\alpha \cdot \Delta\dot{\mathbf{r}}_{in}^\alpha < 0$. Now this relative ingoing velocity must be handled such that $\mathbf{n}^\alpha \cdot \Delta\dot{\mathbf{r}}_{out}^\alpha \geq 0$. The relation will be modelled by the normal restitution coefficient ζ_n^α:

$$\mathbf{n}^\alpha \cdot \Delta\dot{\mathbf{r}}_{out}^\alpha = -\zeta_n^\alpha \mathbf{n}^\alpha \cdot \Delta\dot{\mathbf{r}}_{in}^\alpha \tag{3.4}$$

Note that the tangential restitution must be given to be able to handle colliding contacts, $\zeta_n^\alpha \geq 0$. Furthermore, two optional restitution coefficients will be available. This is a restitution coefficient for tangential relative velocity and angular velocity respectively. The tangential relation will be modelled by the restitution coefficient ζ_t^α:

$$\Delta\dot{\mathbf{r}}_{out}^\alpha - \mathbf{n}^\alpha \cdot \Delta\dot{\mathbf{r}}_{out}^\alpha = -\zeta_t^\alpha(\Delta\dot{\mathbf{r}}_{in}^\alpha - \mathbf{n}^\alpha \cdot \Delta\dot{\mathbf{r}}_{in}^\alpha) \qquad \text{if} \qquad e_t^\alpha = 1 \tag{3.5}$$

and the angular relation by the angular restitution coefficient ζ_a^α:

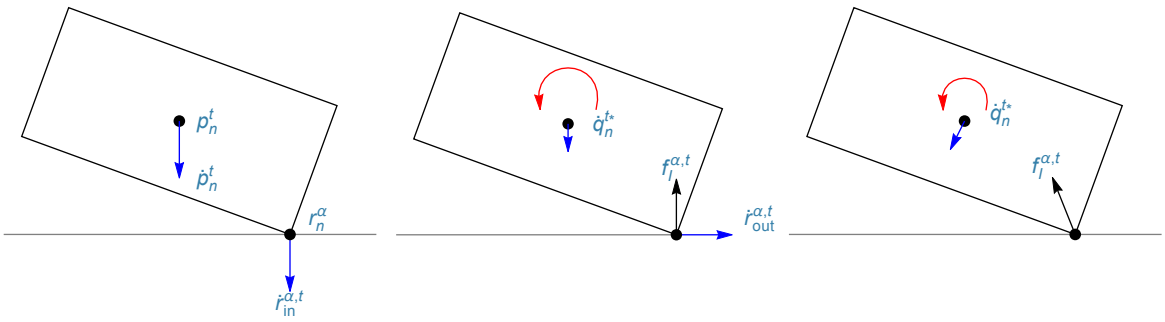
$$\mathbf{n}^\alpha \cdot \Delta\mathbf{w}_{out}^\alpha = -\zeta_a^\alpha \mathbf{n}^\alpha \cdot \Delta\mathbf{w}_{in}^\alpha \qquad \text{if} \qquad e_a^\alpha = 1 \tag{3.6}$$

The binary variables e_t^α and e_a^α are used to enable the restitution in tangential and angular directions respectively. To summarise, the complete restitution model is given by the tuple:

$$M_R^\alpha = (\zeta_n, \zeta_t, \zeta_a, e_t, e_a)^\alpha \tag{3.7}$$

Typically the ζ parameters are fixed constant values, but the user can implement more detailed models where ζ is a function of the positions and velocities of the contact. The tuple is recalculated in the beginning of each step of the simulation loop based on the initial positions and velocities, $\mathbf{q}_n$ and $\dot{\mathbf{q}}_n$. Definition of custom restitution models is the topic of section 6.5. The restitution is often given as a single parameter, ζ^α, in which case the model $M_R^\alpha = (\zeta, 0, 0, 0, 0)^\alpha$ is assumed.

For illustration of the tangential restitution's effect, consider the two different models in figure 3.2. Initially a cylinder is falling, as shown in figure 3.2a, causing a penetrating relative velocity. If the collision is modelled as shown in figure 3.2b, the normal restitution is zero and the tangential restitution is disabled. An impulse, $\mathbf{f}_I^{\alpha,t}$, will be applied in the normal direction such that the normal relative velocity is reduced to zero. The result is an outgoing



(a) Initial configuration (b) $M_R^\alpha = (0, 0, 0, 0, 0)$ (c) $M_R^\alpha = (0, 0, 0, 1, 0)$

Figure 3.2: Illustration of the effect of tangential restitution modelling.

relative velocity that is tangential to the normal. This will cause the box to slide across the surface. If the collision, on the other hand, is modelled as in figure 3.2c, the tangential restitution is enabled and set to zero. This causes an impulse to be applied such that there is neither normal or tangential relative velocity after the collision. This gives an effect similar to sticking friction. In this example, we find that the model that looks most physically correct is the model that does not include tangential restitution. Notice that in this example there is no ingoing relative velocity. Hence any choice of ζ_t will give the same result. In general ζ_t and ζ_a can be both positive and negative.

Our definition of a restitution model allows each contact to have a separate model. In practice, each object is assigned to a collision category label by a map, $L_C : V \to \Sigma_C$. An overall collision model, C, is then determined from a collision map, M_C, that maps $\{L_C(S^\alpha), L_C(T^\alpha)\} \to C$. Notice that the collision map provides the ability to statically specify overall collision handling between bodies by assigning them to generic collision classes. Each collision model then allows collision modelling of each individual contact $M_R^\alpha = C(E^\alpha, \mathbf{q}_n, \dot{\mathbf{q}}_n)$ dynamically during execution. The model C provides the values in equation 3.7.

This concludes the modelling of a contact in collision. In figure 3.2, an example was given of a single collision between two bodies. Notice that many bodies can be in contact simultaneously, and each pair of bodies in contact will in general have multiple simultaneous contacts or constraints. Because of this, determining the impulses is not necessarily a trivial task. Determining the impulses that respects the restitution models is the main topic of section 3.5.

3.5 Collision Solver

The collision solver is responsible for dealing with new colliding contacts, $\mathbb{E}_i^{new}$. It is known in advance that all contacts in this set have a relative colliding velocity. The collision solver must ensure that no colliding or persistent contact have colliding relative velocity after collision

handling. Furthermore, all bilateral velocity constraints must also be satisfied after collision handling. In those cases it can be impossible to satisfy this post-condition. in which case the simulator should exit with an error. This will allow the user to take explicit action by choosing a different collision solver, with the drawbacks that might pose.

Collisions are modelled with a velocity-based restitution coefficient relating the velocity before and after collision. This is a quite simple concept to grasp when two bodies collide in a single point. Unfortunately, the system is rarely that simple, and realistic modelling of collisions in complex systems is not yet well-understood. Algorithm 1 shows the overall collision handling system. The input to the algorithm is the current positions and velocities

Algorithm 1 Overall collision solver method

1: **function** CollisionSolver($\mathbf{q}_n$,$\dot{\mathbf{q}}_n$,G_n^S)
2: $[\mathbb{E}_i^{new}]_n \leftarrow \mathbb{E}_i^{new}(G_n^S)$ ▷ Extract colliding contact set
3: $\mathbb{E}_{collide} \leftarrow$ CollidingVelocity($\mathbf{q}_n$,$\dot{\mathbf{q}}_n$,$[\mathbb{E}_i^{new}]_n$) ▷ Extra safety check
4: $\dot{\mathbf{q}}_n^* \leftarrow$ DoCollisions($\mathbb{E}_{collide}$,$\mathbf{q}_n$,$\dot{\mathbf{q}}_n$,S_n^G) ▷ The core collision handling
5: $[G_n^S]^* \leftarrow$ SetAllLabelsKnown(G_n^S) ▷ L_E^i labels changed to persistent contacts
6: **return** $\dot{\mathbf{q}}_n^*$ and $[G_n^S]^*$
7: **end function**

of all bodies in the system as well as the state of the body-constraint graph, G_n^S. This state includes the persistent and colliding contacts, and this distinction is important in collision handling. Notice that the body-constraint graph, G, is not specified explicitly as this graph is considered constant during a single simulation step. In line 2 the set of new contacts is extracted. Only the ones having relative velocities that cause them to collide will used for resolving collisions in line 4. In this case, it is checked explicitly that all new contacts are in fact colliding. This is an unnecessary step, but checking the pre-conditions is very useful in debugging. As a final step the edge labels are changed, in line 5, for all new contacts, such that they become persistent contacts after collisions have been resolved. This disables rollback and enables contacts forces and friction for all contacts known at this point. The result of the collision solver is new body velocities, $\dot{\mathbf{q}}_n^*$, and an updated state of the body-constraint graph, $[G_n^S]^*$.

The DoCollisions function is implemented in different collision solvers that are available by default. Also, the plugin structure, see section 6.5, allows custom definition of a collision solver if the user wants to handle the collisions in a certain way. It is, however, very important for the overall simulation loop that the post-conditions are satisfied, such that no contacts have relative colliding velocity after collision handling. In the following, four different velocity-based collision solvers are presented. A brief overview is shown in table 3.3. Notice that all solvers depend on the *simultaneous solver* (section 3.5.1). If the simultaneous solver is used on its own, it solves for every contact and constraint in the system simultaneously. This tends to give an averaged impulse response that is not very realistic. The other three solvers are sequential solvers. They have the common problem that they might fail to converge. The simultaneous solver will not experience this issue, hence it can be used as a resolver

Solver	Depends on	Advantages	Disadvantages
Simultaneous		Robust (works always) Solver of last resort	Not realistic
Single Pair	Simultaneous	Multiple simultaneous contacts	2-body collisions only
Chain	Single	N-body collisions in chain Multiple contacts for body-pairs	No circular contacts
Hybrid (default)	Simultaneous	In-between Simultaneous & Chain Robust Propagation when possible	Simultaneous behaviour Can fail in certain scenarios

Table 3.3: Overview of the available velocity-based collision solvers.

of last resort if other solvers fail. The *single pair solver* (section 3.5.3) handles only 2-body collisions. If there are multiple contacts and/or constraints between two bodies, these are solved simultaneously. The solver is mostly provided as a intermediate solver, which is used by the chain solver. The *chain solver* (section 3.5.4) handles collisions in chains by propagation of impulses. In simple assembly tasks there are only few objects, and typically the contacts and constraints form a chain. The solver solves for body-pairs in shift, using the single-pair solver to handle multiple contacts between a body-pair. Finally, the *hybrid solver* (section 3.5.2) handles more complex configurations and uses the simultaneous solver to solve for many object-pairs in simultaneous collision. It will, however, try to reduce the number of bodies handled simultaneously, making it a sort of hybrid between the chain and simultaneous solvers.

3.5.1 The Simultaneous Solver

The simultaneous solver is a basic building block for the other solvers, but can be used on its own to solve for all collisions in the system. The drawback of solving all collisions simultaneously is that the collision response is known to become less realistic. The method will on the other hand make the collision handling very robust. In algorithm 2, the overall method is shown. In line 2, the problem is first broken down into the smallest possible

Algorithm 2 The overall simultaneous collision solver

```
1: function DoCollisions(𝔼_collide, q_n, q̇_n, G_n^S)
2:     graphs ← DynamicComponentsSpringDecoupling(G_n^S)
3:     for all SG in graphs do                    ▷ Each can be handled in parallel
4:         if 𝔼_i^new(SG) ≠ ∅ then
5:             q̇_n^* ← CollideSimultaneous(𝕍_D(SG), 𝔼_SG, q_n, q̇_n)    ▷ Algorithm 3
6:         end if
7:     end for
8:     return q̇_n^*
9: end function
```

dynamic components. Notice that we consider springs a decoupling factor as springs can not carry impulses. Each component can then be handled in parallel as the impulses and body velocities in one component are completely independent from the other components.

Notice, in line 4, that only components that have colliding contacts in them are handled at this point. In line 5 we solve for the impulses in the component to find updated velocities for the dynamic bodies. Before considering algorithm 3, the mathematics is first considered.

Impulse-based Dynamics Formulation

For each dynamic body, $V_D^i \in \mathbb{V}_D$, with contacts and constraints, $E^\beta \in \mathbb{E}_e(V_D^i) \cup \mathbb{E}_i(V_D^i)$, the net linear and angular impulse cause a change in the body velocities:

$$\begin{aligned} \sum_\beta \mathbf{f}_I^{\beta,i} &= m_i(\dot{\mathbf{p}}_n^{i*} - \dot{\mathbf{p}}_n^i) \\ \sum_\beta \mathbf{t}_I^{\beta,i} &= \mathbf{I}_n^i(\mathbf{w}_n^{i*} - \mathbf{w}_n^i) - \mathbf{I}_n^i \sum_\beta (\mathbf{r}_n^{\beta,i} - \mathbf{p}_n^i) \times \mathbf{f}_I^{\beta,i} \end{aligned} \tag{3.8}$$

In equation 3.3, the velocity of a point on the surface was expressed in terms of the ingoing velocities. Now, the outgoing velocity of a constraint or contact point, E^α, can be expressed as a function of the applied impulse:

$$\begin{bmatrix} \dot{\mathbf{r}}_{out}^{\alpha,i} \\ \mathbf{w}_{out}^{\alpha,i} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{p}}_n^{i*} + \mathbf{w}_n^{i*} \times (\mathbf{r}_n^{\alpha,i} - \mathbf{p}_n^i) \\ \mathbf{w}_n^{i*} \end{bmatrix} = \mathbf{a}_I^{\alpha,i} + \sum_\beta \mathbf{B}_I^{\alpha,\beta,i} \begin{bmatrix} \mathbf{f}_I^{\beta,i} \\ \mathbf{t}_I^{\beta,i} \end{bmatrix} \tag{3.9}$$

Here the vector $\mathbf{a}_I^{\alpha,i}$ is the independent term with the outgoing velocities if no impulses are applied. The matrix $\mathbf{B}_I^{\alpha,\beta,i}$ is a mapping to the velocity in contact E^α caused by an impulse applied at a contact, E^β. This linear relationship can easily be constructed by insertion of $\dot{\mathbf{q}}_n^{i*}$ and $\mathbf{w}_n^{i*}$ from equation 3.8.

In section 3.4, the restitution model for a contact was presented. For a unified treatment of constraints and contacts, the model is stated slightly more generic as:

$$\begin{bmatrix} \Delta\dot{\mathbf{r}}_{out}^{\alpha} \\ \Delta\mathbf{w}_{out}^{\alpha} \end{bmatrix} = \mathbf{H}_\zeta^\alpha \begin{bmatrix} \Delta\dot{\mathbf{r}}_{in}^{\alpha} \\ \Delta\mathbf{w}_{in}^{\alpha} \end{bmatrix} = \Delta\mathbf{a}_I^\alpha + \sum_\gamma \mathbf{B}_I^{\alpha,\gamma} \begin{bmatrix} \mathbf{f}_I^\gamma \\ \mathbf{t}_I^\gamma \end{bmatrix} \tag{3.10}$$

Notice that we are now summing over all constraint and contact edges in the system, $E^\gamma \in \mathbf{E}_e \cup \mathbf{E}_i$, instead of summing over only the bodies in connection with either the source or target body. For impulses not influencing E^α, the $\mathbf{B}$ matrix is simply set to zero. Notice that special attention must be taken if $S^\alpha \notin \mathbb{V}_D$ or $T^\alpha \notin \mathbb{V}_D$ in which case the outgoing velocity, $\dot{\mathbf{r}}_{out}^{\alpha,i}$, is given directly from the velocity of the corresponding kinematic or static body. For a constraint edge $E^\alpha \in \mathbb{E}_e$, the matrix $\mathbf{H}_\zeta^\alpha$ is zero. If on the other hand $E^\alpha \in \mathbb{E}_i$ then $\mathbf{H}_\zeta^\alpha$ is a matrix representing the contact restitution model.

Notice, that there are six constraints in 3.10, meaning that the model is too generic for both constraints and contacts. In these equations, both the velocity conditions and the linear and angular impulse are expressed in world coordinates. Instead of deriving a wide range of different bilateral constraint types, as well as a unilateral contact type, a more generic approach is chosen. For each edge, E^α, an edge selection matrix is used. Two rotations are specified in local coordinates relative to the parent body $V^s = S(E^\alpha)$. One orthonormal

basis specifies the basis directions for linear motion, $\mathbf{R}^{\alpha}_{s,lin}$, while another specifies the basis directions for angular motion, $\mathbf{R}^{\alpha}_{s,ang}$. Let $\mathbf{G}^{\alpha}_{w}$ denote the generalised rotation matrix for the constraint:

$$\mathbf{G}^{\alpha}_{w} = \begin{bmatrix} \mathbf{R}^{s}_{w}\mathbf{R}^{\alpha}_{s,lin} & \mathbf{0} \\ \mathbf{0} & \mathbf{R}^{s}_{w}\mathbf{R}^{\alpha}_{s,ang} \end{bmatrix} \tag{3.11}$$

We now define the entities in equation 3.10 using a local coordinate system:

$$\mathbf{B}^{\alpha,\gamma}_{I,loc} = [\mathbf{G}^{\alpha}_{w}]^{-1}\mathbf{B}^{\alpha,\gamma}_{I}\mathbf{G}^{\gamma}_{w} \tag{3.12}$$

$$\mathbf{H}^{\alpha}_{\zeta,loc} = [\mathbf{G}^{\alpha}_{w}]^{-1}\mathbf{H}^{\alpha}_{\zeta}\mathbf{G}^{\alpha}_{w} \tag{3.13}$$

$$\mathbf{i}^{\gamma}_{I,loc} = [\mathbf{G}^{\gamma}_{w}]^{-1}\mathbf{i}^{\gamma}_{I} \tag{3.14}$$

$$\Delta\mathbf{a}^{\alpha}_{I,loc} = -[\mathbf{G}^{\alpha}_{w}]^{-1}\Delta\mathbf{a}^{\alpha}_{I} \tag{3.15}$$

$$\Delta\mathbf{v}^{\alpha}_{I,loc} = [\mathbf{G}^{\alpha}_{w}]^{-1}\begin{bmatrix} \Delta\dot{\mathbf{r}}^{\alpha}_{in} \\ \Delta\mathbf{w}^{\alpha}_{in} \end{bmatrix} \tag{3.16}$$

where $\mathbf{i}$ denotes the generalised impulse composed of $\mathbf{f}_I$ and $\mathbf{t}_I$. The problem can then be written in local coordinates as:

$$\sum_{\gamma}\mathbf{B}^{\alpha,\gamma}_{I,loc}\mathbf{i}^{\gamma}_{I,loc} = \mathbf{H}^{\alpha}_{\zeta,loc}\Delta\mathbf{v}^{\alpha}_{I,loc} - \Delta\mathbf{a}^{\alpha}_{I,loc} \tag{3.17}$$

Now for each of the six constraint directions, the constraint can be enabled or disabled. A diminished version of the equation system, where the rows and columns corresponding to the degrees of freedom are removed, is constructed:

$$\sum_{\gamma}\widehat{\mathbf{B}^{\alpha,\gamma}_{I,loc}}\widehat{\mathbf{i}^{\gamma}_{I,loc}} = \widehat{\Delta\mathbf{c}^{\alpha}_{I,loc}} - \widehat{\Delta\mathbf{a}^{\alpha}_{I,loc}} \tag{3.18}$$

$$\widehat{\Delta\mathbf{c}^{\alpha}_{I,loc}} = \widehat{\mathbf{H}^{\alpha}_{\zeta,loc}}\widehat{\Delta\mathbf{v}^{\alpha}_{I,loc}} \tag{3.19}$$

If for instance a revolute joint is modelled, the last row is removed from all $\mathbf{B}^{\alpha,x}$ matrices. This removes the conditions on velocity in this degree of freedom. Similarly, columns are removed from the $\mathbf{B}^{x,\gamma}$ matrices, making sure that there can be no impulse in this direction. This is a very generic definition that allows us to freely select the orthogonal directions we want to constrain. The directions can be specified independently for the linear and angular directions as we have different basises for the linear and angular part respectively. Notice also that this generic definition fits to contacts as well. If the orthonormal basis is defined with an axis along the contact normal, the others will lie in the tangential directions. Usually the two tangent vectors are then rotated around the normal to align one of the vectors with the direction of motion. If we put together all constraints, the constraint solver must solve the equation system:

$$\begin{bmatrix} \widehat{\mathbf{B}^{1,1}_{I,loc}} & \widehat{\mathbf{B}^{1,2}_{I,loc}} & \cdots & \widehat{\mathbf{B}^{1,\gamma}_{I,loc}} \\ \widehat{\mathbf{B}^{2,1}_{I,loc}} & \widehat{\mathbf{B}^{2,2}_{I,loc}} & \cdots & \widehat{\mathbf{B}^{2,\gamma}_{I,loc}} \\ \vdots & \vdots & \ddots & \vdots \\ \widehat{\mathbf{B}^{\alpha,1}_{I,loc}} & \widehat{\mathbf{B}^{\alpha,2}_{I,loc}} & \cdots & \widehat{\mathbf{B}^{\alpha,\gamma}_{I,loc}} \end{bmatrix} \begin{bmatrix} \widehat{\mathbf{i}^{1}_{I,loc}} \\ \widehat{\mathbf{i}^{2}_{I,loc}} \\ \vdots \\ \widehat{\mathbf{i}^{\gamma}_{I,loc}} \end{bmatrix} = \begin{bmatrix} \widehat{\Delta\mathbf{c}^{1}_{I,loc}} \\ \widehat{\Delta\mathbf{c}^{2}_{I,loc}} \\ \vdots \\ \widehat{\Delta\mathbf{c}^{\alpha}_{I,loc}} \end{bmatrix} - \begin{bmatrix} \widehat{\Delta a^{1}_{I,loc}} \\ \widehat{\Delta a^{2}_{I,loc}} \\ \vdots \\ \widehat{\Delta a^{\alpha}_{I,loc}} \end{bmatrix} \tag{3.20}$$

The solution of this linear equation system gives the diminished impulse vectors. These must then be expanded appropriately to get full 6D impulse vector. The vector is then converted back to world coordinates:

$$\mathbf{i}_I^\gamma = \mathbf{G}_w^\gamma \mathbf{i}_{I,loc}^\gamma \tag{3.21}$$

Finally, equation 3.8 can be used to determine the new velocities for bodies by summing all impulses acting on each body.

Now we have the required mathematical framework for definition of the algorithm 3. In

Algorithm 3 Handle a component with simultaneous collisions

```
1:  function CollideSimultaneous(𝕍_D,𝔼,q_n,q̇_n)
2:      Divide in two sets 𝔼_e and 𝔼_i
3:      Determine the number of inequalities N_in ← |𝔼_i|.
4:      Construct B̂_{I,loc}, Δĉ_{I,loc} and Δâ_{I,loc} with N_in inequalities last.   ▷ See equation 3.20
5:      Determine the number of equalities N_e ← N − N_in
6:      î_{I,loc} ← LinearImpulseSolver(B̂_{I,loc},Δĉ_{I,loc} − Δâ_{I,loc},N_e)        ▷ Algorithm 4
7:      Expand diminished vector î_{I,loc} → i_{I,loc}
8:      Rotate to global coordinates i_{I,loc} → i_I                                  ▷ See equation 3.21
9:      q̇*_n ← ApplySolution(i_I,𝕍_D,𝔼_i,𝔼_e,q_n,q̇_n)                                 ▷ See equation 3.8
10:     return q̇*_n
11: end function
```

line 4, the equation system 3.20 is constructed. Notice that the equation system is ordered such that the first rows contain the constraints $\mathbb{E}_e$ followed by the entries related to the tangent restitution of the contact in $\mathbb{E}_i$. Then the angular restitution follows and finally the normal restitution. The same applies for the columns where the last columns are related to the normal impulse in contacts. This is important for the LinearImpulseSolver, which is called in line 6 to solve the diminished equation system. After solving for the impulses the diminished vector is transformed back into a full impulse vector in world coordinates in lines 7 and 8. Finally, the impulses are used to calculate the velocity change in line 9, and this updates velocity of all bodies in the component returned in line 10.

Solution of Equation System

We will continue to discuss the solution of the linear equation system. The equation system 3.20 is now written in a slightly simplified notation:

$$\mathbf{A}\mathbf{f} = \mathbf{b} \tag{3.22}$$

The impulses are then simply found as a solution to this linear equation system. Unfortunately, one has to consider the problem where the impulse, in unilateral contacts, acts in a direction that effectively pulls the bodies together. Due to the way the contact constraint is defined, this happens if the corresponding normal impulse becomes positive, $[\mathbf{f}^{in}]_i > 0$. Here $\mathbf{f}^{in}$ is the sub-part of $\mathbf{f}$ that holds all contact normal impulses. To solve this issue there are

two possible approaches. Either the problem is solved as a traditional LCP or as a minimisation problem. In our case we will do the latter. We want the smallest possible impulses that solve the constraints and causes the inequality conditions to be true. This results in the following constrained minimisation problem:

$$\min \frac{1}{2}\mathbf{f}^T\mathbf{f}, \qquad \mathbf{Af} = \mathbf{b}, \qquad \mathbf{f}^{in} \leq \mathbf{0} \tag{3.23}$$

The Singular Value Decomposition (SVD) of the matrix $\mathbf{A}$ is used to get the decomposed matrices $\mathbf{U}$, $\mathbf{W}$ and $\mathbf{V}$, all belonging to $\Re^{N\times N}$. The tolerance, ϵ^{abs}_{SVD}, is used to reduce the rank. Diagonal values of $\mathbf{W}$ below this tolerance is set to zero. The tolerance is specified using the following relation:

$$\epsilon^{abs}_{SVD} = N \max_{i=1}^{N}(W_{ii}) \cdot \epsilon^{rel}_{SVD} \tag{3.24}$$

Hence the rank is decreased if the eigenvalues are too small relative to the largest eigenvalue. Note, however, that the rank, K, should never become bigger than $6|V_D|$.

By doing SVD and limiting the rank of the system, we can now work with the problem in the range and the null space respectively. The part of the impulse vector that lies in the null-space should be changed such that it minimises the impulses while keeping them negative for the inequalities. The part of the impulse vector that lies in the range should be changed such that it solves the equation system, but at the same time also fulfils the objectives for the impulses. This is a trade-off, but primary focus should be ,on solving the equality equations. Assuming we have the SVD decomposition of matrix $\mathbf{A}$, the method works as follows. First, define $K \times N$ matrices $\mathbf{U}_s, \mathbf{V}_s$ by the first K columns in $\mathbf{U}$ and $\mathbf{V}$ respectively and define $\mathbf{W}_s$ as the upper left $K \times K$ matrix in $\mathbf{W}$. Let further $\mathbf{V}_s^{\perp}$ be the last $N - K$ columns in $\mathbf{V}$. For an arbitrary $\mathbf{f}$, we now obtain relations between $\mathbf{f}$ and the coordinates of $\mathbf{f}$ in the orthonormal basis given by the columns of $\mathbf{V}_s$ and $\mathbf{V}_s^{\perp}$ as follows:

$$\begin{aligned}
\mathbf{f} &= \mathbf{V}_s\phi_s + \mathbf{V}_s^{\perp}\phi_s^{\perp} && (3.25)\\
\phi_s &= \mathbf{V}_s^T\mathbf{f} && (3.26)\\
\phi_s^{\perp} &= [\mathbf{V}_s^{\perp}]^T\mathbf{f} && (3.27)\\
\mathbf{f}^T\mathbf{f} &= \phi_s^T\phi_s + [\phi_s^{\perp}]^T\phi_s^{\perp} && (3.28)\\
\mathbf{Af} &= \mathbf{U}_s\mathbf{W}_s\phi_s && (3.29)
\end{aligned}$$

We now introduce slack variables, $\mathbf{s}_B$, $\mathbf{s}_X$, and our new coordinates to rewrite our optimisation problem 3.23 as:

$$\begin{aligned}
&\min \frac{1}{2}\{\phi_s^T\phi_s + [\phi_s^{\perp}]^T\phi_s^{\perp}\} && (3.30)\\
&\mathbf{U}_s\mathbf{W}_s\phi_s - \mathbf{s}_B = \mathbf{b} && (3.31)\\
&\mathbf{V}_s\phi_s + \mathbf{V}_s^{\perp}\phi_s^{\perp} - \mathbf{s}_X = \mathbf{0} && (3.32)
\end{aligned}$$

Although this problem seems more complicated, it is actually easier to devise an algorithm for. Assume that we have a desired change, $\mathbf{\Delta s}_B$ and $\mathbf{\Delta s}_X$, for the slack variables. We then

obtain an (overdetermined) system of $N + K$ linear equations for the corresponding change of the K coordinates in $\mathbf{\Delta}\phi_s$:

$$\mathbf{\Gamma U}_s\mathbf{W}_s\mathbf{\Delta}\phi_s = \mathbf{\Gamma\Delta s}_B \tag{3.33}$$

$$\mathbf{\Delta}\phi_s = \mathbf{V}_s^T\mathbf{\Delta s}_X \tag{3.34}$$

Notice that the matrix $\mathbf{\Gamma}$ is a weighting matrix that we will come back to later. The weighting matrix allows control of the trade-off between satisfying the equality constraints and the objectives for the impulse. The normal equations (Moore-Penrose inverse) are:

$$\mathbf{\Delta}\phi_s = (\mathbf{J}_s + \mathbf{W}_s\mathbf{U}_s^T\mathbf{\Gamma}^2\mathbf{U}_s\mathbf{W}_s)^{-1}\left[\mathbf{W}_s\mathbf{U}_s^T\mathbf{\Gamma}^2\Delta\mathbf{s}_B + \mathbf{V}_s^T\mathbf{\Delta s}_X\right] \tag{3.35}$$

where $\mathbf{J}_s$ is an identity matrix. The optimal change, $\mathbf{\Delta}\phi_s^{\perp}$, is:

$$\mathbf{\Delta}\phi_s^{\perp} = [\mathbf{V}_s^{\perp}]^T\mathbf{\Delta s}_X \tag{3.36}$$

The values of $\mathbf{\Delta s}_B$ and $\mathbf{\Delta s}_X$ are chosen as the negative gradient, $\mathbf{g}(\mathbf{s}_B, \mathbf{s}_X)$, to the objective:

$$f(\mathbf{s}_B, \mathbf{s}_X) = \frac{1}{2}\sum_{i=1}^{N}[(\mathbf{s}_B)_i]^2 + \alpha[(\mathbf{s}_X)_i]^2 + \frac{1}{2}\sum_{i=N_e+1}^{N_e+N_i}\max[(\mathbf{s}_X)_i, 0]^2 \tag{3.37}$$

The first term is supposed to ensure that all equality constraints are satisfied. The second α-term tries to keep the impulses small. The value of α should be rather small. When we are close to the optimum, we set α to zero to ensure that the constraints are satisfied at the expense of the impulses being less distributed. The third term ensures that the inequality conditions for the normal contact impulses are satisfied.

Choice of Weights, Γ, and Efficiency Concerns

In general, it will be more important to ensure that constraints and restitution models are satisfied than negativity of the impulse. For constraints the outgoing velocity should be zero, no matter what impulse it requires. To put more weight on velocity-level equalities, it is proposed to choose a rather high, fixed value for the corresponding value for Γ_{ii}. As an example $\gamma^{max} = 100$ can be used. Now, it is proposed that for an inequality constraint, i, a high fixed value for Γ_{ii} is used as well, but only as long as the outgoing normal velocity of the contact is less than modelled by the restitution model. If $(s_B)_i > 0$, the outgoing velocity is higher than modelled by the restitution model. In this case we lower the weight to focus on the impulse objectives:

$$\Gamma_{ii} = \gamma^{max}\min\{1, \|\Delta s_B\|/(s_B)_i\} \tag{3.38}$$

This expression will not exceed γ^{max} and will decrease to zero as $(s_B)_i$ gets bigger or the overall velocity errors in the system decrease to zero. This weighting method will cause the iterative method to converge faster.

The weighting matrix causes the invertion of a matrix as shown in equation 3.35. This is potentially a computationally heavy operation as it must be performed at each step of

the algorithm. It requires a recomputation of $\Gamma^2(U_s W_s)$ and in particular $(W_s U_s^T)(\Gamma^2 U_s W_s)$, which is an $N \times N$ matrix multiplication. Far from the solution or if convergence is fast enough, we recommend to use γ^{max} for all i, in which case we obtain the simplified solution:

$$\mathbf{\Delta}\phi_s = (\mathbf{J}_s + (\gamma^{max})^2 \mathbf{W}_s^2)^{-1} \left[(\gamma^{max})^2 \mathbf{W}_s \mathbf{U}_s^T \mathbf{\Delta s}_B + \mathbf{V}_s^T \mathbf{\Delta s}_X \right] \tag{3.39}$$

Note that $\mathbf{J}_s + (\gamma^{max})^2 \mathbf{W}_s^2$ is a diagonal matrix, which is easily calculated and inverted.

The Linear Impulse Solver Algorithm

Now, consider the algorithm 4, which summarises our approach. The input is $\mathbf{A}$ and $\mathbf{b}$ as already discussed. Notice that the first N_e rows and columns of $\mathbf{A}$ are equalities with no limits for the corresponding impulses. The remaining rows and columns are associated to contacts and normal impulses that have certain objectives. Initially, a SVD operation is performed in

Algorithm 4 Iterative solver for determining the solution to a linear equation system

Require: $\mathbf{A} \in \Re^{N \times N}$, $\mathbf{b} \in \Re^{N}$, $N > 0$, $0 \leq N_e \leq N$

1: **function** LINEARIMPULSESOLVER($\mathbf{A}$,$\mathbf{b}$,N_e)
2: $(\mathbf{U}, \mathbf{\Sigma}, \mathbf{V^T}) \leftarrow$ SINGULARVALUEDECOMPOSITION($\mathbf{A}$,ϵ_{SVD}^{rel}) ▷ See equation 3.24
3: Determine the rank K from $\mathbf{\Sigma}$
4: **if** $K = N$ **then** ▷ If $\mathbf{A}$ is full rank try to solve directly
5: $\mathbf{s}_X \leftarrow \mathbf{V}\mathbf{W}^{-1}\mathbf{U}^T\mathbf{b}$
6: **if** $\max_{i=N_e+1}^{N} (\mathbf{s}_X)_i \leq 0$ **then** ▷ Only if $\mathbf{f}^{in} \leq 0$ is satisfied
7: **return** $\mathbf{s}_X$
8: **end if**
9: **end if**
10: Construct $\mathbf{U}_s$, $\mathbf{\Sigma}_s$, $\mathbf{V}_s$, $\mathbf{V}_s^T$, $\mathbf{V}_s^\perp$ and $\mathbf{V}_s^{\perp T}$ based on K
11: Calculate constants $\mathbf{G}_s = \mathbf{U}_s\mathbf{W}_s$ and $\mathbf{G}_s^T$
12: $\phi \leftarrow \mathbf{0} \in \Re^K$, $\phi^\perp \leftarrow \mathbf{0} \in \Re^{N-K}$
13: $\mathbf{s}_B \leftarrow -\mathbf{b}$, $\mathbf{s}_X \leftarrow \mathbf{0} \in \Re^N$
14: $k \leftarrow 0$
15: **repeat**
16: **if** $k > 0$ and $e < \alpha_t$ **then**
17: $\alpha \leftarrow 0$
18: **end if**
19: Compute $(\mathbf{\Delta s}_B, \mathbf{\Delta s}_X)^T = -\mathbf{g}(\mathbf{s}_B, \mathbf{s}_X)$ ▷ See equation 3.37
20: **if** $\max_{i=N_e+1}^{N_e+N_i} (\mathbf{s}_X)_i \leq 0$ **then**
21: Calculate $\mathbf{B} = (\mathbf{J}_s + (\gamma^{max})^2\mathbf{W}_s^2)^{-1}$
22: $\mathbf{\Delta}\phi_s = \mathbf{B}\left[(\gamma^{max})^2\mathbf{G}_s^T\mathbf{\Delta s}_B + \mathbf{V}_s^T\mathbf{\Delta s}_X\right]$ ▷ See equation 3.39
23: **else**
24: Compute Γ
25: Calculate $\mathbf{B} = (\mathbf{J}_s + \mathbf{G}_s^T\mathbf{\Gamma}^2\mathbf{G}_s)^{-1}$
26: $\mathbf{\Delta}\phi_s = \mathbf{B}\left[\mathbf{G}_s^T\mathbf{\Gamma}^2\mathbf{\Delta s}_B + \mathbf{V}_s^T\mathbf{\Delta s}_X\right]$ ▷ See equation 3.35
27: **end if**
28: $\mathbf{\Delta}\phi_s^\perp = [\mathbf{V}_s^\perp]^T\mathbf{\Delta s}_X$ ▷ See equation 3.36
29: $\mathbf{s}_B^{(k+1)} = \mathbf{s}_B^{(k)} + \mathbf{G}_s\mathbf{\Delta}\phi_s$ ▷ See equation 3.33
30: $\mathbf{s}_X^{(k+1)} = \mathbf{s}_X^{(k)} + \mathbf{V}_s\mathbf{\Delta}\phi_s + \mathbf{V}_s^\perp\mathbf{\Delta}\phi_s^\perp$ ▷ See equation 3.34
31: $k \leftarrow k + 1$
32: $e \leftarrow \|\mathbf{\Delta}\phi_s\| + \|\mathbf{\Delta}\phi_s^\perp\|$
33: **until** $k > k_{max}$ or $e < \epsilon_{obj}$
34: **return** $\mathbf{s}_X$
35: **end function**

line 2. Based on the chosen tolerance, the rank is determined in line 3 as K. Lines 4 to 9 attempt to solve the problem directly if the problem has full rank. Notice that this requires

that the normal impulse in contacts must be negative. If non-negative impulses are found, the algorithm continues to line 10 to 14, where initial variables are constructed. The main loop in lines 15 to 34 shows how we iteratively moves toward a solution that minimises our objective function. In line 16 to 18 the α parameter is set to zero if we get close to the solution. The gradient of the objective function is determined in line 19. If no normal restitution models are fulfilled yet, the lines 21 and 22 are executed using the simplified solution, where all weights are equal. This gives an efficient evaluation when we are far from a fulfilment of the restitution model. If, on the other hand, we are close to a solution, a weight is calculated in line 24 causing the more heavy computations in line 25 and 26. In lines 28 to 30 the remaining values are calculated, and the slack variables are updated. These slack values are now closer to an optimisation of the objective function. In lines 31 and 32 the iteration count and error measure is updated. Line 33 iterates until the change in the slack variables becomes too small or the maximum iteration count is reached. In line 34 the found impulses are returned.

This concludes the walk-through of the simultaneous solver. It is the most complex solver that implements all the heavy math for handling of collisions. The remaining collision solvers will be much more simple as they use the same functions as have been presented in this subsection. Notice that in section 3.10 a method is presented for solving constraint forces. The method will be very similar to the approach presented in this section.

3.5.2 The Hybrid Solver

The hybrid solver has better support for simultaneous collisions. The algorithm still prefers handling to handle collisions independently if possible. Contrary to the chain solver, cyclic dependencies can be handled by using the simultaneous solver. In worst case the hybrid solver can solve the entire system with the simultaneous solver. In algorithm 5 the details of the solver is shown. In line 2, the system is initially broken down into independent sub-problems if possible. This is similar in all solvers. In lines 4 to 39 each component is handled in parallel. Two sets are used to control the propagation of collisions. The initially colliding contacts are added to set $\mathbb{C}$ in line 5, and all contacts and constraints are initially added to a disabled set, $\mathbb{D}$, in line 6. As the impulses propagate, the contacts will be enabled. In line 7 the initial collisions are enabled to initialise the propagation. The loop in lines 9 to 38 runs until there are no more colliding contacts in $\mathbb{C}$ or the maximum iteration count is reached. First, the solver will try to extract a second level of independent subcomponents. This is done based on the current collision set and will allow us to use the maximum possible sequential handling. In line 12 and 13 each of these subcomponents are processed using the simultaneous solver. Notice that all edges in the component are enabled, allowing them to become colliding in the next iteration. After handling each sub-component, the colliding set is cleared in line 15, and a new colliding set is constructed in lines 16 to 36. In line 17 to 22 the relative velocity of each contact and constraint is checked. If a contact or constraint requires an impulse to be applied at this point, the edge is added to the colliding set in line 25. However, this happens only if the edge has been enabled previously. If the edge is disabled, we need to check if it can be enabled in line 27 to 33. Enabling the edge requires that it connects to a node that already has an enabled edge.

By using this strategy we take care that collisions propagate and that collisions can be handled simultaneously if required. This concludes the section on collision solvers. Note that it is in general not fully understood how to model collisions of many bodies with many simultaneous contacts. We find that the hybrid solver models collisions in a realistic way, while at the same time being robust if many simultaneous collisions should occur.

Algorithm 5 The hybrid collision solver

function DoCollisions($\mathbb{E}_{collide}$,$\mathbf{q}_n$,$\dot{\mathbf{q}}_n$,G_n^S)
  components ← DynamicComponentsSpringDecoupling(G_n^S)
  $\dot{\mathbf{q}}_n^* \leftarrow \dot{\mathbf{q}}_n$
  for all SG in components **do** ▷ Each can be handled in parallel
    $\mathbb{C} \leftarrow \mathbb{E}_i^{new}(SG)$ ▷ Initially colliding contacts
    $\mathbb{D} \leftarrow \mathbb{E}_e(SG) \cup \mathbb{E}_i(SG)$ ▷ Disabled set
    $\mathbb{D} \leftarrow \mathbb{D} \setminus \mathbb{C}$ ▷ Enable initial collisions
    $i \leftarrow 0$
    while $\mathbb{C} \neq \emptyset$ and $i \leq i_{max}$ **do**
      subcomponents ← ConnectedComponents(SG,$\emptyset$,$\mathbb{C}$)
      for all CC in subcomponents **do** ▷ Can be done in parallel
        $\mathbb{D} \leftarrow \mathbb{D} \setminus \mathbb{E}(CC)$ ▷ Enable all edges for future propagation
        $\dot{\mathbf{q}}_n^* \leftarrow$ CollideSimultaneous($\mathbb{V}_D(CC)$,$\mathbb{E}(CC)$,$\mathbf{q}_n$,$\dot{\mathbf{q}}_n^*$) ▷ See algorithm 3
      end for
      $C \leftarrow \emptyset$
      for all $E^i \in \mathbb{E}(SG)$ **do**
        solve ← false
        if $E^i \in \mathbb{E}_i(SG)$ and $\mathbf{v}_{rel} \cdot \mathbf{n} < -\epsilon_{contact}$ **then**
          solve ← true
        else if $E^i \in \mathbb{E}_e(SG)$ and $\|\Delta\mathbf{v}^i\|_\infty > \epsilon_{\Delta v}$ or $\|\Delta\mathbf{w}^i\|_\infty > \epsilon_{\Delta w}$ **then**
          solve ← true
        end if
        if solve **then**
          if $E^i \notin \mathbb{D}$ **then**
            $\mathbb{C} \leftarrow \mathbb{C} \cup E^i$ ▷ Add to collision set
          else
            for all CC in subcomponents **do**
              if $S(E^i) \in \mathbb{V}(CC)$ or $T(E^i) \in \mathbb{V}(CC)$ **then**
                $\mathbb{C} \leftarrow \mathbb{C} \cup E^i$ ▷ Add to collition set
                $\mathbb{D} \leftarrow \mathbb{D} \setminus E^i$ ▷ Enable contact
                break
              end if
            end for
          end if
        end if
      end for
      $i \leftarrow i + 1$
    end while
  end for
  return $\dot{\mathbf{q}}_n^*$
end function

3.5.3 Single Body-Pair Solver

This type of solver is proposed mostly as an utility solver for more complex algorithms. As the name suggests it can handle collisions between only one pair of bodies, (V^s, V^t), at a time. It does, however, handle an arbitrary mix of constraints and contacts between these two bodies. The collision solver implementation is shown in algorithm 6. In lines 2 to 9 a

Algorithm 6 The single body-pair solver

```
1:  function DoCollisions(𝔼_collide, q_n, q̇_n, G_n^S)
2:      pairs ← ∅
3:      for all contacts E^i in 𝔼_collide do
4:          if {S(E^i), T(E^i)} ∉ pairs then
5:              If S(E^i) ∈ 𝕍_D, ensure that {S^α, T^α} = {S^i, T^i} ∀ E^α ∈ 𝔼(S^i)
6:              If T(E^i) ∈ 𝕍_D, ensure that {S^α, T^α} = {S^i, T^i} ∀ E^α ∈ 𝔼(T^i)
7:              pairs ← pairs ∪{S(E^i), T(E^i)}
8:          end if
9:      end for
10:     for all {P^A, P^B} in pairs do                    ▷ Can be done in parallel
11:         q̇_n^* ← CollideSimultaneous({P^A, P^B}, 𝔼(P^A, P^B), q_n, q̇_n)    ▷ Algorithm 3
12:     end for
13:     return q̇_n^*
14: end function
```

set of body-pairs is constructed. Each colliding contact will cause a body-pair to be added in line 7. Notice that before creation of a collision pair, it is checked in lines 5 and 6 if one of the two bodies has collisions with a third body. If this is the case, the algorithm fails. In lines 10 to 12 each independent body-pair is solved using algorithm 3.

3.5.4 Chain Solver

Collision chains can be handled by the chain solver. Based on the colliding contacts, the solver tries to construct a chain of bodies in contact. Each body-pair is given an index according to the location in the chain. Then even and odd pairs are handled in shifts in an iterative fashion until there are no more collisions to process. The single-pair solver is used underneath to handle each body-pair in parallel. Notice that the chain allows multiple contacts or constraints between a pair of bodies, but there can not be any circular dependencies between bodies. Notice that the chain solver handles all static bodies as a single body. The chain solver can be useful in certain assembly simulations that typically have a few bodies, which come into contact in a chain-like fashion. However, the hybrid solver is more useful in a general setting. Hence we will not consider the chain solver in greater detail.

3.6 Position & Velocity Update

In multi-body dynamics, the motion of rigid bodies can be formulated using different approaches. Often used formulations in classical mechanics are Newtonian, Lagrangian and Hamiltonian formulations. Lagrangian mechanics is based on preservation of the total kinetic and potential energy in a system and is formulated in a minimal set of coordinates. Newtonian mechanics works directly in Cartesian space and finds forces and torques acting between bodies in full coordinates. In our case, we find the maximal coordinates representation ideal as we need to study manipulation actions with many degrees of freedom, as well as contacts and friction. The Newton-Euler equations were given in equation 2.2 in a slightly abstract notation. In this section, we will consider the following set of Newton-Euler equations:

$$\begin{aligned} \ddot{\mathbf{p}}^i &= m_i^{-1}\mathbf{f}^i(\mathbf{p}, \dot{\mathbf{p}}, \boldsymbol{\Omega}, \mathbf{w}^i, S_c) \\ \dot{\mathbf{w}}^i &= \mathbf{I}(\boldsymbol{\Omega}^i)^{-1}\left(\mathbf{t}^i(\mathbf{p}, \dot{\mathbf{p}}, \boldsymbol{\Omega}, \mathbf{w}^i, S_c) - \mathbf{w}^i \times \mathbf{I}(\boldsymbol{\Omega}^i)\mathbf{w}^i\right) \end{aligned} \tag{3.40}$$

where

$\mathbf{p}$ is the position of center of mass in global coordinates. Hence $\dot{\mathbf{p}}$ and $\ddot{\mathbf{p}}$ are the corresponding velocity and acceleration.

$\boldsymbol{\Omega}$ is the orientation in global coordinates. It is custom to represent this entity as a quaternion. In this section we simply consider it a rotation matrix in $\Re^{3\times3}$.

$\mathbf{w}$ is the angular velocity in global coordinates. Hence $\dot{\mathbf{w}}$ is the angular acceleration.

m is the mass.

$\mathbf{I}$ is the inertia given in a global reference frame around the center of mass. Note that for a rigid body, the inertia will be constant in a local reference frame, but the inertia given in global reference frame is a function of $\boldsymbol{\Omega}$.

$\mathbf{f}^i$ is the net force acting on the body. This is dependent on the positions and velocities of all bodies in the system (consider for instance a damped spring force). Furthermore, the force can not be expected to be continuous as contacts might change in a discontinuous manner. The contacts also cause friction, which is modelled using hysteresis. Hence the force has state S_c, which will be discussed in section 3.9.

$\mathbf{t}^i$ is the net torque acting on the body. It has the same characteristics as the net force.

The net force and torque will be further composed of a range of different forces acting on the body:

$$\mathbf{f}^i = \mathbf{f}^{ext,i} + \sum_{\beta\in\mathbb{E}(\mathbb{V}_D^i)} \mathbf{f}_{app}^{\beta,i} + \mathbf{f}^{\beta,i} \tag{3.41}$$

$$\mathbf{t}^i = \mathbf{t}^{ext,i} + \sum_{\beta} \mathbf{t}_{app}^{\beta,i} + \mathbf{t}^{\beta,i} + \Delta\mathbf{r}^{\beta,i} \times \left(\mathbf{f}_{app}^{\beta,i} + \mathbf{f}^{\beta,i}\right) \tag{3.42}$$

where

$\mathbf{f}^{ext,i}$ **and** $\mathbf{t}^{ext,i}$ are forces and torques applied directly at the body due to gravity or controller inputs.

$\mathbf{f}^{\beta,i}_{app}$ **and** $\mathbf{t}^{\beta,i}_{app}$ are forces and torques applied directly at an edge. This is, for instance, spring forces, viscous friction or controller inputs for constraints.

$\mathbf{f}^{\beta,i}$ **and** $\mathbf{t}^{\beta,i}$ are forces and torques acting at a constraint or contact to satisfy velocity and friction constraints. These are the product of the contact and constraint solver, which will be discussed in section 3.10.

Determination of the forces and torques is not the topic in this section. Here it will simply be assumed that $\mathbf{f}^i$ and $\mathbf{t}^i$ are known. Determining the motion of a body then involves integration of the Newton-Euler equations. Two different integration schemes will be used to integrate the motion, namely explicit Euler and Heun's method. These are first and second-order methods respectively, and it is the goal to use the second-order method where possible, and the first order method otherwise.

3.6.1 Explicit Euler Integration Scheme

Explicit Euler integrates positions and velocities independently based on only the previous positions and velocities. The positions are:

$$\begin{aligned} \mathbf{p}^i_{n+1} &= \mathbf{p}^i_n + h\dot{\mathbf{p}}^i_n \\ \mathbf{\Omega}^i_{n+1} &= \mathbf{R}_{EAA}(h\mathbf{w}^i_n)\mathbf{\Omega}^i_n \end{aligned} \tag{3.43}$$

where $\mathbf{R}_{EAA}$ is a rotation matrix corresponding to a rotation around the equivalent angle axis (EAA). The velocities are integrated as:

$$\begin{aligned} \dot{\mathbf{p}}^i_{n+1} &= \mathbf{p}^i_n + hm_i^{-1}\mathbf{f}^i_n \\ \mathbf{w}^i_{n+1} &= \mathbf{w}^i_n + h[\mathbf{I}^i_n]^{-1}(-\mathbf{w}^i_n \times \mathbf{I}^i_n\mathbf{w}^i_n + \mathbf{t}^i_n) \end{aligned} \tag{3.44}$$

3.6.2 Heun's Integration Scheme

First, the Heun scheme makes a prediction of the velocity:

$$\begin{aligned} \dot{\mathbf{p}}^p_{n+1} &= \dot{\mathbf{p}}_n + hm^{-1}\mathbf{f}_n \\ \mathbf{w}^p_{n+1} &= \mathbf{w}_n + h\mathbf{I}^{-1}(-\mathbf{w}_n \times \mathbf{I}\mathbf{w}_n + \mathbf{t}_n) \end{aligned} \tag{3.45}$$

The position is then updated based on this prediction:

$$\begin{aligned} \mathbf{p}_{n+1} &= \mathbf{p}_n + \frac{h}{2}(\dot{\mathbf{p}}_n + \dot{\mathbf{p}}^p_{n+1}) \\ \mathbf{\Omega}_{n+1} &= \mathbf{R}_{EAA}(\frac{h}{2}(\mathbf{w}_n + \mathbf{w}^p_{n+1}))\mathbf{\Omega}_n \end{aligned} \tag{3.46}$$

Now, a new, corrected velocity is found:

$$\begin{aligned}\dot{\mathbf{p}}_{n+1} &= \dot{\mathbf{p}}_n + \frac{h}{2} m^{-1}(\mathbf{f}_n + \mathbf{f}_{n+1}) \\ \mathbf{w}_{n+1} &= \mathbf{w}_n + \frac{h}{2}([\mathbf{I}_n^i]^{-1}\mathbf{t}_n + [\mathbf{I}_{n+1}^i]^{-1}\mathbf{t}_{n+1}) \\ &\quad - \frac{h}{2}([\mathbf{I}_n^i]^{-1}\mathbf{w}_n \times \mathbf{I}_n^i\mathbf{w}_n + [\mathbf{I}_{n+1}^i]^{-1}\mathbf{w}_{n+1} \times \mathbf{I}_{n+1}^i\mathbf{w}_{n+1})\end{aligned} \tag{3.47}$$

This is a second-order predictor-corrector method that uses an explicit Euler step as a predictor for velocity. A second-order method will be expected better to be able to model gravity, the gyroscopic term and the behaviour of springs than the explicit Euler method.

3.7 Rollback

Adaptive time-stepping is used to resolve collisions at the time of impact. The approach used poses certain requirements on contact detection. This will be discussed in section 3.12. As some contact detection schemes do not function well if bodies are in collision, care must be taken to avoid any penetration for these object pairs. However, other detection schemes can handle penetrations and give reliable estimations of penetration depth. It is desirable that both cases can be handled by the rollback algorithm while being able to exploit the penetration depth information when available. Hence rollback will be done in two stages: a broad-phase stage, which uses a bisection strategy to avoid penetrations, and a narrow-phase stage, which allows better strategies for resolving the time of impact under the assumption that penetration depth can be estimated.

3.7.1 Broad-Phase Rollback

The broad-phase rollback method is shown in algorithm 7. For each step of the iterative

Algorithm 7 Broad-phase rollback method

Require: No initial penetrations for body-pairs using contact strategies which does not handle penetrations.

1: **function** RollbackBroadPhase(h,Int,$\mathbf{q}_n$,$\dot{\mathbf{q}}^*_n$,$\mathbf{f}^*_n$)
2: **for** $i = 1$ to $i^{max}_{rollback}$ **do** ▷ Limit in case precondition is not satisfied
3: $s = \frac{h}{2^{i-1}}$
4: $(\mathbf{q}^*_{n+1}, \dot{\mathbf{q}}^P_{n+1}) \leftarrow$ UpdatePosition(s,Int,$\mathbf{q}_n$,$\dot{\mathbf{q}}^*_n$,$\mathbf{f}^*_n$) ▷ See equation 3.43 and 3.46
5: **if** not MaxPenetrationExceeded($\mathbf{q}^*_{n+1}$) **then**
6: **return** Δt, $\mathbf{q}^*_{n+1}$ and $\dot{\mathbf{q}}^P_{n+1}$
7: **end if**
8: **end for**
9: **end function**

Ensure: No penetrations for the relevant body-pairs.

method, the step-size, Δt, is divided by two. This happens in line 3 until no penetrations occur or the maximum number of iterations is reached. In the latter case the engine will exit with a failure, which will indicate that either the number of maximum iterations is reached or the pre-condition was not satisfied (there was a penetration initially). In line 4, the position of all bodies is integrated using an integration scheme, Int. In line 5, a collision check is performed for the newly integrated positions. Note that only object pairs, that use a contact strategy that can not handle penetrations are tested. Other strategies will never result in penetration in this test. In section 3.12, the contact strategies will be discussed further. Note that in general the collision check, performed in line 5, can be done efficiently as it is an overlap test and not all contacts need to be generated. A standard PQP collision test is used currently.

3.7.2 Narrow-Phase Rollback

Resolving the time of impact is a root-finding problem, and it is proposed to use the higher order Ridder's method to resolve the time of impact. To illustrate the procedure, an example is given in figure 3.3. In this example, the distance between two different body-pairs is shown over a timestep from t_1 to t_3. There is no initial collisions at time t_1, but after integrating the system to time t_3, the body-pairs penetrate. For now it is assumed that the distance and depth can be evaluated at all times within the interval. Note that in the case where broad-phase rollback is used, the penetration depth can not be determined. Hence, it is difficult to use a root search method to determine the time of impact. However, in this case there will be a contact layer around the object, and it will be assumed that penetrations can be determined if they are smaller than the size of the contact layer. The penetration depth will in this case refer to the size of the penetration of the contact layer and not of the object itself. This means that rollback will be performed such that objects collide exactly where the

contact layers touch.

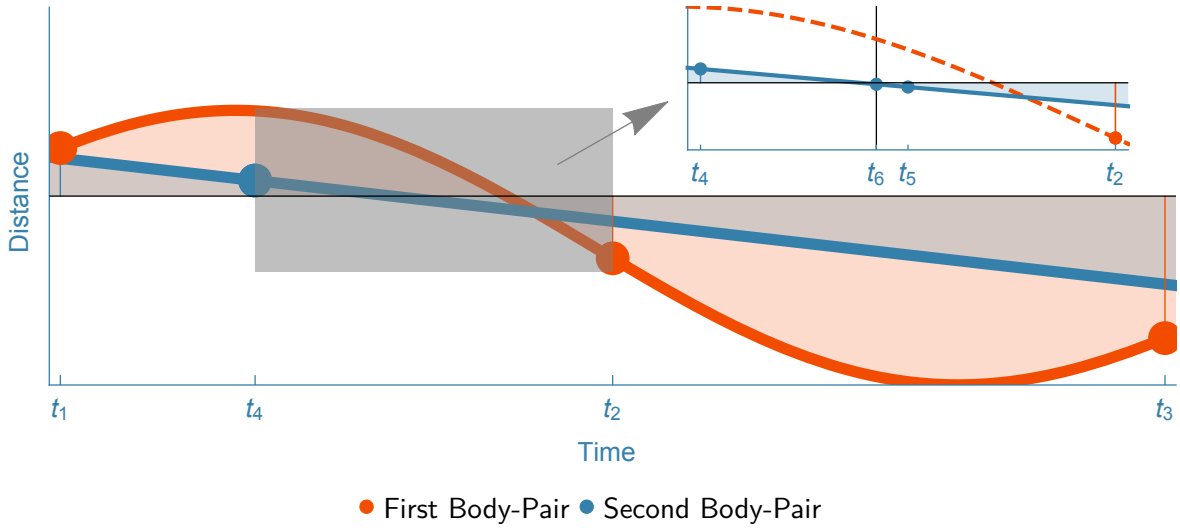


Figure 3.3: Resolving the time of impact with multiple body-pairs close to collision.

Now, consider the penetration depth of a contact as a function of the step-size, $d^\alpha(t)$. For new contacts we then have:

$$\begin{aligned} d^\alpha(t_n) < & \quad -\epsilon_{layer} - \epsilon_{rollback} \\ d^\alpha(t_{n+1}) > & \quad -\epsilon_{layer} + \epsilon_{rollback} \end{aligned} \tag{3.48}$$

where $\epsilon_{rollback}$ is the desired tolerance, and ϵ_{layer} is the size of the contact layer. The goal is to determine the step-size $t_n < s < t_{n+1}$, such that $d^\alpha(s) = -\epsilon_{layer}$.

In figure 3.3, it is seen that two different body-pairs collide during the time interval. In this case it is important to resolve the first impact. Notice that no matter what distance the two body-pairs have at times t_1 and t_3, it is impossible to infer which of the body-pairs causes the first collision. It is reasonable to use the heuristic that the pair with the deepest penetration at time t_3 is most likely to be the pair causing the first collision. In the example the deepest penetration at t_3 happens for the body-pair that does not cause the first impact. A variant of Ridder is proposed that allows switching the body-pair being used for root-finding. Ridder is a two-step method where $t_2 = \frac{t_1+t_3}{2}$ is found first. When the distances have been evaluated at t_2, a new guess for t_4 can be found:

$$t_4 \quad = \quad t_2 + (t_2 - t_1)\text{sign}(d_1 - d_3)\frac{d_2}{\sqrt{d_2^2 - d_1 d_3}} \tag{3.49}$$

The deepest penetrating body-pair at t_3 is used to determine the value for t_4. After evaluation at t_4, there are now four samples stored for each body-pair. The new active pair becomes the deepest penetrating pair at t_4. According to the sign of the distances, the root is then bracketed between t_4 and t_2. At $t_5 = \frac{t_4+t_2}{2}$, the second body-pair is the only one penetrating. Therefore the first body-pair is discarded as we know that the collision can not happen for this pair first. The new guess for t_6 is finally determined from the penetration

depths for the second body-pair only, and we arrive at exactly the first impact point. By storing the deepest penetrations for each object pair instead of a global minimum distance, the algorithm is expected to converge faster. In this example, the former requires 4 evaluations to arrive at an exact solution, where the latter uses 6 iterations.

Some special cases and failures can occur. If for instance a body-pair is suddenly lost, the algorithm will continue using remaining body-pairs (if any). If for instance no distance can be determined for t_4 for the first body-pair, t_4 will be calculated for the remaining body-pair. In this case, there is no longer a guarantee that the first collision is found. In the method, it is assumed that the distance function crosses zero only once during the time-step. If it crosses multiple times, there is no guarantee that the first collision is found. Finally, it might happen that the distance is already within the desired precision at time t_1. This is not a problem in the Ridder method as such, but we will get back to this issue when discussing the narrow-phase rollback and its relation to contact detection.

In the given example, two different object-pairs in collision were considered. Note that the exact same problem can occur with a single body-pair as well if there can be multiple contacts. If the bodies have a persistent contact while rotating relative to each other, a new collision might occur, and that must be resolved to avoid penetrations. An example of this is a cylinder falling onto a plane due to gravity. If the restitution is zero, the cylinder will experience a collision at one end, and that will cause an angular velocity of the cylinder and a persistent contact between the cylinder and the plane. Shortly after, the other end of the cylinder will collide with the plane. The angular velocity can become quite big, and this shows that it is not enough to consider the global minimum distance between two bodies. The local distances in each contact must be considered. In this section, the search method has been described. It will be the responsibility of the narrow-phase rollback algorithm to make sure that persistent contacts are not included in the distances used in the search method. The purpose of the narrow-phase rollback method is to resolve the exact time of impact when collisions occur. This is important to avoid penetrations and to allow accurate collision handling. The method is outlined in algorithm 8. The input to the narrow-phase rollback algorithm is the time-step, Δt, the integration scheme, Int, and the initial positions, $\mathbf{q}_n$. $\mathbf{q}^*_{n+1}$ are the positions after time-step Δt. Furthermore, a set of updated contacts, $\tilde{\mathbb{E}}_n$, and the newly detected contacts, $\mathbb{E}^{new}_{n+1}$, is used. Notice the tracking sets $\mathbb{T}_n$ and $\mathbb{T}_{n+1}$. These are the tracking information associated to each contact at time t_n and t_{n+1} respectively. First, in line 2 to 5, we check if rollback is required at all. If there are no contacts with a penetrating velocity or the penetration depth is too small, we simply take a complete timestep, Δt. In line 6, the contacts at time t_{n+1} are tracked backwards in time to determine their location at time t_n. In line 7, only the contacts that were classified as new contacts are used. These correspond to $\mathbb{E}^{new}_{n+1}$, just at an earlier point in time. If the set $\mathbb{N}$ is empty, the tracking must have failed to determine the location of the contact at t_n. In this case, the rollback ends in line 9 by returning the complete time-step, Δt.

Algorithm 8 Narrow-phase rollback method

Require: No initial penetration

1: **function** RollbackNarrowPhase(Δt,Int,$\mathbf{q}_n, \mathbf{q}^*_{n+1}, \dot{\mathbf{q}}^*_n, \dot{\mathbf{f}}^*_n, \tilde{\mathbb{E}}_n, \mathbb{E}^{new}_{n+1}, \mathbb{T}_n, \mathbb{T}_{n+1}$)
2: $\mathbb{E}_p \leftarrow$ PenetratingVelocity($\mathbb{E}^{new}_{n+1}$,$\mathbf{q}^*_{n+1}$,$\epsilon_{collisionvel}$)
3: **if** $\mathbb{E}_p \in \emptyset$ or MaximumPenetration($\mathbb{E}_p$) $> \epsilon_{rollback}$ **then**
4: **return** Δt
5: **end if**
6: $\begin{bmatrix}\tilde{\mathbb{E}}^{\leftarrow}_n \\ \mathbb{T}^{\leftarrow}_n\end{bmatrix} \leftarrow$ TrackContacts($\mathbf{q}_n$,$\mathbb{T}_{n+1}$)
7: $\mathbb{N} \leftarrow \mathbb{E}^{new}_i(\tilde{\mathbb{E}}^{\leftarrow}_n, \mathbb{T}^{\leftarrow}_n)$ ▷ Extract new contacts only
8: **if** $\mathbb{N} \in \emptyset$ **then**
9: **return** Δt ▷ Tracking failed
10: **else**
11: **if** MinimumDistance($\mathbb{N}$) $< \epsilon_{rollback}$ **then** ▷ Initial distance within precision
12: Restart the simulation loop with additional contacts $\mathbb{N}$
13: **return** 0
14: **else**
15: $\mathbb{D} \leftarrow$ ConstructSample($t_n, \mathbb{N}_n$) $\cup$ ConstructSample($t_{n+1}, \mathbb{N}_{n+1}$)
16: **repeat**
17: $s \leftarrow$ SearchMethod($\mathbb{D}$) ▷ Such as Ridder's Method
18: $(\mathbf{q}^*_{n+1}, \dot{\mathbf{q}}^P_{n+1}) \leftarrow$ UpdatePosition(s,Int,$\mathbf{q}_n$,$\dot{\mathbf{q}}^*_n$,$\mathbf{f}^*_n$) ▷ Eq 3.43 and 3.46
19: $\begin{bmatrix}\mathbb{E}^{\leftarrow}_s \\ \mathbb{T}^{\leftarrow}_s\end{bmatrix} \leftarrow$ TrackAndFindContacts($\mathbf{q}^*_{n+1}$,$\mathbb{T}_{n+1}$)
20: $\mathbb{N}_s \leftarrow \mathbb{E}^{new}_i(\mathbf{E}^{\leftarrow}_s, \mathbb{T}^{\leftarrow}_s)$
21: **if** $|$MinimumDistance($\mathbb{N}_s$)$| > \epsilon_{rollback}$ **then**
22: $\mathbb{D} \leftarrow \mathbb{D} \cup$ ConstructSample($t_s, \mathbb{N}_s$)
23: **else**
24: **return** s, $\mathbf{q}^*_{n+1}$ and $\dot{\mathbf{q}}^P_{n+1}$
25: **end if**
26: **until** iterations $< i^{rollback}_{max}$
27: **end if**
28: **end if**
29: **end function**

Ensure: No penetrations after timestep

In the lines 11 to 13, a special case is handled. We might be unlucky that the contact depth at time t_n was within the rollback tolerance. This means that there was a new contact, which was never handled in the collision solver. In this case we must restart the simulation loop with the additional initial colliding contact. In the lines 15 to 26, the core rollback root search is performed. Two samples are constructed in line 15 for the contact sets at t_n and t_{n+1}. In line 17, a search method, such as Ridder's method, is called with the two initial

samples to determine a new guess for the time-step, s. In line 18, we make an update of the body positions, and when using these new positions a new backward tracking is performed in line 19. In line 20, the new contacts are extracted. If the closest contact is within the tolerance of rollback, we are done and return the new value for s in line 24. Otherwise, a new sample is constructed in line 22, and a new iteration is performed.

3.7.3 Overall Rollback & Position Update

Now, we have looked at the two rollback methods for broad-phase and narrow-phase rollback. In algorithm 9, the full combined position update and rollback method is shown. The rollback module is responsible for finding the correct size of the time step, $0 \leq s \leq \Delta t$, such that collisions are handled with the correct time of impact. As part of the process of finding s, the positions are also integrated forward with the given time, s. Dependent on the integration scheme used, Int, the algorithm might also give a prediction of the velocities at $t_n + s$ as output. In line 2, the broad-phase part is performed. This gives an initial reduction of the

Algorithm 9 The full combined *Position Update & Rollback Method*

Require: No initial penetration

1: **function** PositionUpdateWithRollback(Δt,Int,$\mathbf{q}_n, \dot{\mathbf{q}}_n^*$,$\mathbb{T}_n$)

2: $\begin{bmatrix} s \\ \mathbf{q}_{n+1}^* \\ \dot{\mathbf{q}}_{n+1}^P \end{bmatrix} \leftarrow$ RollbackBroadPhase(Δt,Int,$\mathbf{q}_n, \dot{\mathbf{q}}_n^*, \dot{\mathbf{f}}_n^*$) ▷ Algorithm 7

3: SanityCheckPositions($\mathbf{q}_{n+1}^*$,$|WS|_{max}$) ▷ Check that bodies are within workspace.

4: $\begin{bmatrix} \tilde{\mathbb{E}}_n \\ \mathbb{E}_{n+1}^{new} \\ \mathbb{T}_{n+1} \end{bmatrix} \leftarrow$ TrackAndFindContacts($\mathbf{q}_{n+1}^*$,$\mathbb{T}_n$)

5: **if** $\mathbb{E}_{n+1}^{new} \notin \emptyset$ **then**

6: $\begin{bmatrix} s \\ \mathbf{q}_{n+1}^* \\ \dot{\mathbf{q}}_{n+1}^P \end{bmatrix} \leftarrow$ RollbackNP(s,Int,$\mathbf{q}_n, \mathbf{q}_{n+1}^*, \dot{\mathbf{q}}_n^*, \dot{\mathbf{f}}_n^*, \tilde{\mathbb{E}}_n, \mathbb{E}_{n+1}^{new}, \mathbb{T}_n, \mathbb{T}_{n+1}$) ▷ Algorithm 8

7: **end if**

8: **return** s, $\mathbf{q}_{n+1}^*$ and $\dot{\mathbf{q}}_{n+1}^P$

9: **end function**

Ensure: No penetrations after timestep

timestep from Δt to s if there are hard penetrations, which only occur if contact strategies that cannot handle penetrations, are used. This yields a new set of positions, which will be given a sanity check to ensure that the engine stops if the system explodes. A negative value for $|WS|_{max}$ disables this check. In line 4, the contacts at the new position are determined by a call to the contact detector. The tracking set, $\mathbb{T}_n$, allows the contact detector to update the contacts, which were known at t_n, to time t_{n+1}. These are persistent contacts, and they are already known to not be colliding. Furthermore, the tracking set allows classification of new and potentially colliding contacts. If the contact detector does not classify any contacts as

colliding, the method is done and returns in line 8. If new colliding contacts were determined, the narrow-phase rollback is used in line 6, which will determine the exact time of impact.

As can be seen, the entire system for adaptive time-stepping puts certain requirements on the contact detection system, like tracking and classification of contacts as colliding or non-colliding. These requirements are difficult in practice, and much of the complexity of our engine design lies in the increased demands on contact generation. In section 3.12, we will get back to this topic and discuss algorithms, that work together with our rollback scheme. Even though the complexity of tracking contacts can make contact algorithms quite complex, we strongly believe that it is important to have a concept of stateful contacts, which has temporal coherence over multiple timesteps.

3.8 Correction

Due to errors in integration, small contact and constraints will be introduced in each time-step. Over time, constraints will drift apart if this issue is not addressed. Note also that a contact point on an object in general will move across the surface during the time-step. This means that even though the integration is perfect, the object curvature might cause the normal constraints to be violated after taking a time-step. Often, this is solved by using a penalty force where a force proportional to the positional error is used. This is also referred to as Baumgarte stabilisation. In our case, error correction by projection will be used. Each object will be moved and rotated such that it respects the constraints. The uncorrected position, $\mathbf{p}_0^i$, and orientation, $\mathbf{\Omega}_0^i$, will be corrected to $\mathbf{p}^i$ and $\mathbf{\Omega}^i$ respectively:

$$\mathbf{p}^i = \mathbf{p}_0^i + \Delta\mathbf{p}^i \tag{3.50}$$

$$\mathbf{\Omega}^i = R_{EAA}(\Delta\mathbf{w}^i)\mathbf{\Omega}_0^i \tag{3.51}$$

After correction the constraint configuration is:

$$\mathbf{r}^{\alpha,i} = \mathbf{p}_0^i + \Delta\mathbf{p}^i + R_{EAA}(\Delta\mathbf{w}^i)\left(\mathbf{r}_0^{\alpha,i} - \mathbf{p}_0^i\right) \tag{3.52}$$

$$\mathbf{\Omega}^{\alpha,i} = R_{EAA}(\Delta\mathbf{w}^i)\mathbf{\Omega}_0^{\alpha,i} \tag{3.53}$$

where $\mathbf{\Omega}_0^{\alpha,i} = \mathbf{\Omega}_0^i\mathbf{R}_{ang}^{\alpha,i}$. Using the approximation $\mathbf{R}_{EAA}(\Delta\mathbf{w}^i) \approx \mathbf{J} + [\Delta\mathbf{w}^i]^\times$, the position can be approximated as:

$$\mathbf{r}^{\alpha,i} \approx \mathbf{r}_0^{\alpha,i} + \Delta\mathbf{p}^i + \Delta\mathbf{w}^i \times \left(\mathbf{r}_0^{\alpha,i} - \mathbf{p}_0^i\right) \tag{3.54}$$

Using the constraint $\mathbf{r}^{\alpha,i} - \mathbf{r}^{\alpha,j} = \mathbf{0}$, the following equation is obtained for the linear correction:

$$\mathbf{r}_0^{\alpha,j} - \mathbf{r}_0^{\alpha,i} \approx \Delta\mathbf{p}^i - \Delta\mathbf{p}^j - \left(\mathbf{r}_0^{\alpha,i} - \mathbf{p}_0^i\right)^\times \Delta\mathbf{w}^i + \left(\mathbf{r}_0^{\alpha,j} - \mathbf{p}_0^j\right)^\times \Delta\mathbf{w}^j \tag{3.55}$$

Then we consider the angular correction. Here the correction constraint becomes $[\mathbf{\Omega}^{\alpha,j}]^{-1}\mathbf{\Omega}^{\alpha,i} = \mathbf{J}$. This requires the following to be true:

$$EAA\left(\mathbf{\Omega}_0^{\alpha,j}[\mathbf{\Omega}_0^{\alpha,i}]^{-1}\right) = EAA\left(R_{EAA}(-\Delta\mathbf{w}^j)R_{EAA}(\Delta\mathbf{w}^i)\right) \tag{3.56}$$

Again, a linearisation is performed:

$$EAA\left(\mathbf{\Omega}_0^{\alpha,j}[\mathbf{\Omega}_0^{\alpha,i}]^{-1}\right) \approx \Delta\mathbf{w}^i - \Delta\mathbf{w}^j \tag{3.57}$$

Note that this linearisation is exact if $\Delta\mathbf{w}^i$ and $\Delta\mathbf{w}^j$ are parallel. Unfortunately, it is not possible to make such assumptions in general.

It is then possible to pose the problem as a linear equation system:

$$\mathbf{A}\mathbf{\Delta}x = \mathbf{b} \tag{3.58}$$

where $\mathbf{x}$ is composed of the vectors $\Delta\mathbf{p}^i$ and $\Delta\mathbf{w}^i$. If the number of rigid bodies is K, the vector $\mathbf{x}$ has length $6K$. Correction is done for constraints and contacts only in directions that have corresponding velocity constraints.

Now, consider a tight-fitting scenario. In this case, the equation system might not have a solution. In this case all errors cannot be corrected simultaneously. By using the SVD to solve for $\Delta\mathbf{x}$, it is possible to find the correction required to best fulfil the constraints. If there are multiple solutions, the least possible correction, $\min\|\mathbf{x}\|$, will be used, and if there are no solutions the SVD will find the solution, which brings the constraints as close as possible to being fulfilled, $\min\|\mathbf{b} - \mathbf{A}\mathbf{x}\|$. In the case where a small contact layer, ϵ_{layer}, is used, correction of the body positions might cause a hard penetration that can not be recovered. For this reason, it is proposed that the problem is extended with a definition on how much the contact can penetrate:

$$\mathbf{A}^e\mathbf{x} \simeq \mathbf{b}^e \tag{3.59}$$

$$\mathbf{A}^i\mathbf{x} \leq \mathbf{b}^i \tag{3.60}$$

$$\tag{3.61}$$

A feasible solution should satisfy that the equality constraints, 3.59, are satisfied within some accuracy, $\epsilon^{in}_{correction}$. The inequality constraints, 3.60, are due to contacts. As most contact detection methods rely on non-penetration, we need to operate with a contact layer of some thickness, d^c, to avoid penetration during the subsequent time step. The values of $\mathbf{b}^i$ are chosen so that there is a distance, d_c, between the bodies at all contacts. Therefore, we must accept a worst case violation, $\epsilon^{in}_{correction} \equiv \epsilon_{rel}d_c$, corresponding to an (user chosen) acceptable penetration of the contact layer. It is also important to find a small displacement, $\mathbf{x}$. Here, we may operate with a worst case positional displacement, δ^p, and a worst case angular displacement, δ^a. We therefore introduce slack variables and propose that a solution is found by solving the optimisation problem:

$$\min\frac{1}{2}\left[\mathbf{x}^T\mathbf{\Delta}^{-2}\mathbf{x} + \alpha\mathbf{s}^T\mathbf{E}^{-2}\mathbf{s}\right] \tag{3.62}$$

$$\mathbf{A}^e\mathbf{x} - \mathbf{s}^e = \mathbf{b}^e \tag{3.63}$$

$$\mathbf{A}^i\mathbf{x} - \mathbf{s}^i \leq \mathbf{b}^i \tag{3.64}$$

where $\mathbf{s} = (\mathbf{s}^e, \mathbf{s}^i)$, $\delta = \text{diag}(\delta^p, \delta^a)$, $\epsilon = \text{diag}(\epsilon^e, \epsilon^i)$ and the scalar parameter α is a relative weight between obtaining small adjustments and constraint accuracy, which is to be examined below. By writing $\mathbf{y} = (\delta^{-1}\mathbf{x}, \sqrt{\alpha}\epsilon^{-1}\mathbf{s})$, $\mathbf{b} = (\mathbf{b}^e, \mathbf{b}^i)$ and:

$$\mathbf{B}(\alpha) = \begin{bmatrix} \mathbf{A}^e\delta & | & \\ -- & & -\frac{1}{\sqrt{\alpha}}\sqrt{\epsilon} \\ \mathbf{A}^i\delta & | & \end{bmatrix} \tag{3.65}$$

The optimisation problem can now be rewritten as:

$$\min \frac{1}{2}\mathbf{y}^T\mathbf{y} \tag{3.66}$$

$$\mathbf{B}(\alpha)\mathbf{y} = \mathbf{b} \tag{3.67}$$

The solution becomes (assuming a non-singular $\mathbf{B}(\alpha)$)

$$\mathbf{y}(\alpha) = \mathbf{B}(\alpha)^T[\mathbf{B}(\alpha)\mathbf{B}(\alpha)^T]^{-1}\mathbf{b} \tag{3.68}$$

Singularity problems can be avoided by selecting the solution through Singular Value Decomposition. For a chosen α, we check the worst case conditions. If a condition is violated due to an adjustment, we decrease α, and if a condition is violated due to a constraint, we increase α. As soon as none of the worst case conditions are violated, we return y. If a condition of both types is violated, we terminate with a message that we are in a configuration where no adjustment can be made. With sensible choices of the worst case conditions, this will never happen in practice.

3.8.1 Linearisation Error

Consider the approximation in equation 3.54. After solving the equation system, it will in general turn out that $\mathbf{r}^{\alpha,i} - \mathbf{r}^{\alpha,j} \neq \mathbf{0}$ due to the approximation. By expressing the $\mathbf{R}_{EAA}(\Delta\mathbf{w}^i)$ matrix, using the Rodriguez formula, the error of linearisation can easily be determined:

$$\begin{aligned} \mathbf{R}_{EAA}(\Delta\mathbf{w}^i) &= \mathbf{J} + \sin\theta_i\mathbf{v}^\times + (1-\cos\theta_i)\mathbf{v}^\times\mathbf{v}^\times & (3.69)\\ &= \mathbf{J} + [\Delta\mathbf{w}^i]^\times + (\sin\theta_i - \theta_i)\mathbf{v}^\times + (1-\cos\theta_i)\mathbf{v}^\times\mathbf{v}^\times & (3.70)\end{aligned}$$

where $\mathbf{v}^i = \frac{\Delta\mathbf{w}^i}{\|\Delta\mathbf{w}^i\|}$ and $\theta^i = \|\Delta\mathbf{w}^i\|$. This gives us the following error:

$$\begin{aligned} \mathbf{r}^{\alpha,j} - \mathbf{r}^{\alpha,i} &= (\sin\theta_i - \theta_i)\mathbf{v}^i \times \Delta\mathbf{r}^i + (1-\cos\theta_i)\mathbf{v}^i \times \left(\mathbf{v}^i \times \Delta\mathbf{r}^i\right) \\ &- (\sin\theta_j - \theta_j)\mathbf{v}^j \times \Delta\mathbf{r}^j - (1-\cos\theta_j)\mathbf{v}^j \times \left(\mathbf{v}^j \times \Delta\mathbf{r}^j\right)\end{aligned}$$

where $\Delta\mathbf{r}^i = \mathbf{r}_0^{\alpha,i} - \mathbf{p}_0^i$.
Now the angle between vectors $\mathbf{v}^i$ and $\Delta\mathbf{r}^i$ is named θ_i, and the size of the error can be written:

$$\begin{aligned} \|\mathbf{r}^{\alpha,j} - \mathbf{r}^{\alpha,i}\| &= \sqrt{(\sin\theta_i - \theta_i)^2 + (1-\cos\theta_i)^2}\|\Delta\mathbf{r}^i\| \\ &+ \sqrt{(\sin\theta_j - \theta_j)^2 + (1-\cos\theta_j)^2}\|\Delta\mathbf{r}^j\|\end{aligned}$$

Now, consider the infinitesimal $\mathbf{W}_{rot}$ approximation:

$$\mathbf{W}_{rot} = \mathbf{v} \sin\theta \tag{3.71}$$

This is related to EAA in the following way:

$$\mathbf{E}AA = \mathbf{v}\theta = \mathbf{W}_{rot}\frac{\theta}{\sin\theta} \tag{3.72}$$

Hence, the $\frac{\theta}{\sin\theta}$ gives the relative error of using $\mathbf{W}_{rot}$ instead of EAA. This can be approximated as:

$$\mathbf{E}AA = \mathbf{W}_{rot}(1 + \frac{\theta^2}{6} + O(\theta)^4) \tag{3.73}$$

Now, the following Taylor linearisation is used:

$$\begin{aligned} \mathbf{W}_{rot}(\mathbf{R}_{EAA}(-\Delta\mathbf{w}^j)\mathbf{R}_{EAA}(\Delta\mathbf{w}^i)) &\approx \Delta\mathbf{w}^i - \Delta\mathbf{w}^j - \frac{1}{2}\Delta\mathbf{w}^i \times \Delta\mathbf{w}^j & (3.74)\\ &+ O([\Delta\mathbf{w}^i]^2, \Delta\mathbf{w}^j) + O(\Delta\mathbf{w}^i, [\Delta\mathbf{w}^j]^2) & (3.75) \end{aligned}$$

If only the second-order terms are considered, the EAA and $\mathbf{W}_{rot}$ error estimates will be the same as the term $\frac{\theta^2}{6}$ can be ignored. The total angular constraint error then becomes:

$$\begin{aligned} \mathbf{E}AA(\mathbf{R}_{EAA}(-\Delta\mathbf{w}^j)\mathbf{R}_{EAA}(\Delta\mathbf{w}^i)) &\approx \Delta\mathbf{w}^i - \Delta\mathbf{w}^j - \frac{1}{2}\Delta\mathbf{w}^i \times \Delta\mathbf{w}^j & (3.76)\\ &+ O([\Delta\mathbf{w}^i]^2, \Delta\mathbf{w}^j) + O(\Delta\mathbf{w}^i, [\Delta\mathbf{w}^j]^2) & (3.77) \end{aligned}$$

Contrary to the linear constraint error, the error of the EAA linearisation can not be expressed exact. Instead, it will be roughly $\frac{1}{2}\Delta\mathbf{w}^i \times \Delta\mathbf{w}^j$. This gives us an idea of the errors that can be expected when doing correction by projection. Notice that all $\|\Delta\mathbf{w}^i\|$ should always be kept small to avoid too large constraint errors.

3.9 Contact Model

Modelling of contacts with friction is often done with linearised Coulomb friction cones in an LCP formulation, as discussed in section 2.2. Figure 3.4 illustrates the the deviation of the effective friction coefficient from the one modelled by the user. Notice that the error of μ increases drastically as the number of vertices decreases. Unfortunately, the use of many vertices causes the corresponding LCP problem to become more complex to solve.

In RWPE, a micro-slip friction model is used instead. This allows us to formulate the dynamics in section 3.10 as a convex problem. Compared to the friction cone approximations, we expect better results as it is possible to make much more advanced friction models when using the micro-slip friction model. The model requires each contact to keep track of the frictional state through multiple steps of the simulation. Hence, robust contact tracking is required for simulation of friction when using micro-slip.

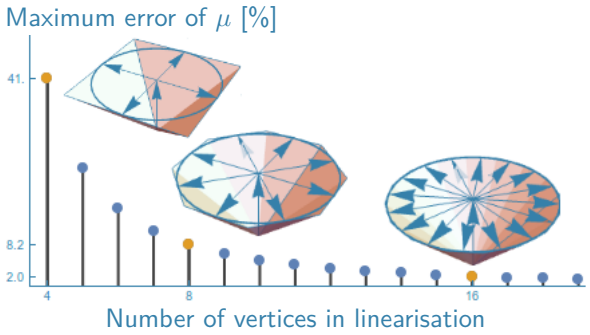


Figure 3.4: Illustration of the expected errors in the Coulomb friction model when different friction cone approximations are used.

Modelling of friction is based on the micro-slip model as described in [90]. Here, a micro-slip regime is used to model the friction when the relative tangential velocity is small. It is argued that in this regime there will be an adhesive force that appear to be a function of the displacement. Hence, the sticking behaviour of static friction is better modelled as a type of spring deformation than as hard velocity constraints. As velocities increase, the gross-sliding regime is entered where the friction is modelled as a function of the relative tangential velocity. In our implementation, the model is extended to three dimensions such that there are two tangential components. The overall friction model is given by:

$$\mathbf{f}_t(\Delta\mathbf{r}_t, \Delta\dot{\mathbf{r}}_t, f_n) = f_g(\Delta\dot{\mathbf{r}}_t, f_n) \cdot \xi(\Delta\mathbf{r}_t, f_n) + c\Delta\dot{\mathbf{r}}_t \tag{3.78}$$

where $\mathbf{f}_t$ is the tangential friction force, and f_n is the normal force. $\Delta\mathbf{r}_t$ and $\Delta\dot{\mathbf{r}}_t$ are the relative position and velocity in the contact. Viscous friction is modelled with the coefficient c. The gross-friction model is modelled with Stribeck friction:

$$f_g(\Delta\dot{\mathbf{r}}_t, f_n) = \left(\mu_c + (\mu_s - \mu_c)e^{\frac{\|\Delta\dot{\mathbf{r}}_t\|}{v_s}}\right) \cdot f_n \tag{3.79}$$

where μ_c is the Coulomb friction coefficient, which is used if the relative velocity is greater than the Stribeck velocity, v_s. The static friction coefficient, μ_s, is used if the relative velocity becomes zero.

Now the ξ is a vector that gives the effective direction of friction and models the micro-slip regime using hysteresis. ξ is a part of the contact state and must be tracked during the contact's existence. The size is $\|\xi\| \leq 1$, and if $\|\xi\| = 1$, friction is in the macro-slip regime where friction will be directed opposite to the relative motion. If $\|\xi\| < 1$, then micro-slip is used to model the sticking phenomena with low relative velocities. The tangential components of ξ are now considered where the $\xi^{\|}$ component works in the tangential direction along the direction of relative motion, and the $\xi^{\perp}$ works perpendicularly to the parallel component in the tangential plane. These are given as:

$$\xi^{\|} = \mathbf{d}\left(1 - \frac{1 - \xi \cdot \mathbf{d}}{\|\Delta\dot{\mathbf{r}}_t\| + e^{-\|\Delta\dot{\mathbf{r}}_t\|}} e^{-h(\gamma\|\Delta\dot{\mathbf{r}}_t\|+r)}\right) \tag{3.80}$$

$$\xi^{\perp} = (\xi - (\xi \cdot \mathbf{d})\mathbf{d})\, e^{-h(\gamma\Delta\dot{\mathbf{r}}_t\|+r)} \tag{3.81}$$

where

$$\mathbf{d} = \begin{cases} \frac{\Delta\dot{\mathbf{r}}_t}{\|\Delta\dot{\mathbf{r}}_t\|} & \text{if} \quad \|\Delta\dot{\mathbf{r}}_t\| \geq \epsilon \\ \frac{\xi_k}{\|\xi_k\|} & \text{if} \quad \|\Delta\dot{\mathbf{r}}_t\| < \epsilon, \|\xi_k\| \geq \epsilon \\ \mathbf{0} & \text{if} \quad \|\Delta\dot{\mathbf{r}}_t\| < \epsilon, \|\xi_k\| < \epsilon \end{cases} \tag{3.82}$$

Here, $\mathbf{d}$ is the direction of relative motion as long as the size of the relative motion is large enough for the direction to be determined with enough precision. If the relative speed is less than the precision, ϵ, then the existing direction of the spring displacement is used instead. At this point, be aware that while the relative tangential motion always lies in the tangential plane, the ξ is a product of the previous relative tangential velocities, and hence it cannot be assumed to lie in the plane. In each iteration, ξ must be projected down to the plane. Equation 3.80 is the main component that models the stick/slip phenomena in the active direction of motion. Equation 3.81 is a fast decaying exponential term with the purpose of reducing the part of ξ, that does not lie in this direction.

Figure 3.5 shows an example of how the friction component will change when the relative velocity is driven in a sinusoidal manner. The Stribeck behaviour is clearly seen: the static

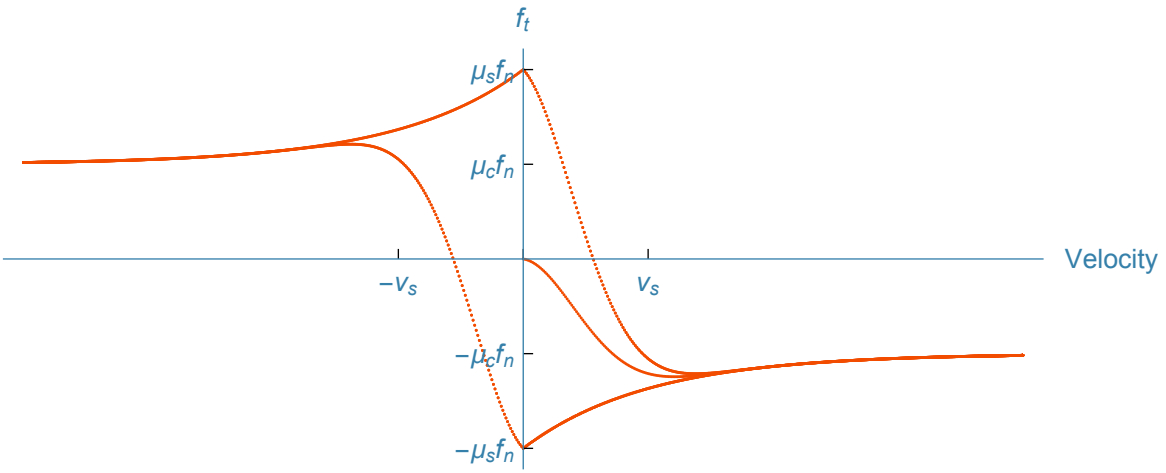


Figure 3.5: The micro-slip friction model used with Stribeck as the gross model

coefficient of friction at zero velocity is μ_s decreasing to μ_c at higher velocities. The micro-slip regime, roughly between $-v_s$ and v_s, shows hysteresis behaviour. At the point where the relative motion changes direction, the friction force will actually keep the same direction as before. Normally the direction of the friction force would change at the same time as the change in motion direction. In this case, however, the change is delayed, and the contact is initially "helped" until it gains speed in the new motion direction.

It is believed that modelling friction, with a model like this, is better than the Coulomb approximations, which are often performed. This method easily supports much more complex models than Coulomb. The drawback is that contact tracking must work reliably for friction to work, and the explicit nature of the friction force can cause oscillations if friction parameters are not chosen wisely. One important thing to notice at this point is the effect of

collision propagation. As the relative velocity will make a continuous jump in this case, the state, ξ, will be cleared when applying impulses.

In general, the user is able to use any custom friction model. The interface is very similar to the interface for the restitution models given in 3.4. The friction model for a contact, α, is given as the tuple:

$$M_F^\alpha = (\mu_t, \mathbf{d}_t, \mu_a, \mathbf{d}_a, e_a)^\alpha \tag{3.83}$$

where e_a allows switching angular friction on and off. μ_t is the tangential friction coefficient to use with tangential friction in the direction $\mathbf{d}_t$, while μ_a is the angular friction coefficient to apply around the vector $\mathbf{d}_a$. Note that $\mathbf{d}_a$ is expected to point in the normal direction, or opposite, allowing the user to model the angular friction with hysteresis as well.

Notice that, a separate model can be used for each contact. In practice, each object is assigned a material label by the map $L_F : V \to \Sigma_F$. An overall friction model, F, is then determined from a collision map, M_F, which maps $\left(L_F^{S(\alpha)}, L_F^{T(\alpha)}\right) \to F$. Similar to the collision map, the friction map provides the ability to statically specify overall friction models between objects of certain materials. Each friction model then allows dynamically modelling of the friction of each individual contact, $M_F^\alpha = F(\alpha, \mathbf{q}_n, \dot{\mathbf{q}}_n)$, during execution. The model, F, provides the value set in equation 3.83.

3.10 Contact & Constraint Resolver

For resting contacts and persistent constraints, a constraint-based solver will be used. In this section the dynamic equations will be formulated, and again we will propose the use of an optimisation method to solve the forces. Much of the theory, presented in this section, should be seen in parallel to the theory for the collision solver in section 3.5. Though it governs different quantities, the structures of the formulations are similar. In this section we strive to solve the inverse dynamics problem, such that the contact and constraint forces can be determined given knowledge about the constrained motion of the system.

3.10.1 Formulation of the Problem

Consider a constraint edge, E^α. The velocity at time t_{n+1} is given as:

$$\begin{bmatrix} \dot{\mathbf{r}}_{n+1}^{\alpha,i} \\ \mathbf{w}_{n+1}^{\alpha,i} \end{bmatrix} = \begin{bmatrix} \dot{\mathbf{p}}_{n+1}^i + \mathbf{w}_{n+1}^i \times (\mathbf{r}_{n+1}^{\alpha,i} - \mathbf{p}_{n+1}^i) \\ \mathbf{w}_{n+1}^i \end{bmatrix} \tag{3.84}$$

The integration schemes discussed in section 3.6 provide expressions for $\dot{\mathbf{p}}_{n+1}^i$ and $\mathbf{w}_{n+1}^i$ in terms of the configuration at time t_n, the time-step, Δt, and the net force and torque acting during the time-step. See for instance the Euler or Heun schemes in equation 3.44 and 3.47. Notice that due to our stepping scheme, the forces and torques at time t_n are already known. Now, we wish to determine the forces and torques at time t_{n+1}. The explicit Euler scheme does not depend on these, but the Heun scheme does. In the Heun scheme the position and

velocity, $\dot{\mathbf{p}}^i_{n+1}$ and $\mathbf{w}^i_{n+1}$, are linearly dependent of the net force and torque, $\mathbf{f}_{n+1}$ and $\mathbf{t}_{n+1}$. Inserting the equations for the net force and torque, 3.41 and 3.42, the expression 3.84 will be linear in $\mathbf{f}^\beta$ and $\mathbf{t}^\beta$. This means that the motion of a constraint on the surface of a body can be written as a linear function of all constraint forces and torques acting on that same body:

$$\begin{bmatrix} \dot{\mathbf{r}}^{\alpha,i} \\ \mathbf{w}^{\alpha,i} \end{bmatrix}^{n+1} = \mathbf{a}^{\alpha,i} + \sum_\beta \mathbf{B}_f^{\alpha,\beta,i}[\mathbf{f}^\beta]^{n+1} + \mathbf{B}_t^{\alpha,\beta,i}[\mathbf{t}^\beta]^{n+1} \tag{3.85}$$

where $\mathbf{a}^{\alpha,i}$ is a term, that gives the velocity of the constraint in case no constraint contact or constraints forces are present in the system. The $\mathbf{B}$ matrices map the forces and torques acting in the system to the velocity change, which they cause in the constraint. Notice the close similarity with 3.9, in which impulses were used to change the velocities instantaneously. The expressions for $\mathbf{a}$ and $\mathbf{B}$ are tedious to derive, so we will not do this here. Instead, be aware that there is a linear relationship between them.

A wide array of different constraints and joint types can now be defined. A fixed constraint will first be considered:

$$\begin{bmatrix} \Delta\dot{\mathbf{r}}^{\alpha} \\ \Delta\mathbf{w}^{\alpha} \end{bmatrix}^{n+1} = \Delta\mathbf{a}^\alpha + \sum_\beta \mathbf{B}_f^{\alpha,\beta}[\mathbf{f}^\beta]^n + \mathbf{B}_t^{\alpha,\beta}[\mathbf{t}^\beta]^n = \Delta\mathbf{v}^\alpha \tag{3.86}$$

where $\Delta\mathbf{v}^\alpha$ is the target relative velocity for the edge E^α. The term $\Delta\mathbf{v}^\alpha$ can for instance be used to model joint motors. Now, ee can write the constraint as:

$$\sum_\beta \mathbf{B}^{\alpha,\beta}\mathbf{i}^\beta_{n+1} = \Delta\mathbf{v}^\alpha - \Delta\mathbf{a}^\alpha \tag{3.87}$$

where $\mathbf{i}^\beta$ is the wrench consisting of contact or constraints force and torque. $\mathbf{B}^{\alpha,\beta}$ defines the contribution from each contact or constraint wrench in the system to one specific contact or constraint, α. Under normal circumstances, the $\mathbf{B}^{\alpha,\beta}$ will be non-zero only if $S^\alpha = S^\beta$, $S^\alpha = T^\beta$, $T^\alpha = S^\beta$ or $T^\alpha = T^\beta$.

In this equation, both the velocity constraint and the wrench are expressed in world coordinates. Instead of deriving a wide range of different bilateral constraint types, the generic approach introduced in section 3.5.1 is used. Completely similar to the approach described there, the equation system is written in local coordinates as a diminished version. Our equation system then has the form:

$$\begin{bmatrix} \widehat{\mathbf{B}^{1,1}_{loc}} & \widehat{\mathbf{B}^{1,2}_{loc}} & \cdots & \widehat{\mathbf{B}^{1,\beta}_{loc}} \\ \widehat{\mathbf{B}^{2,1}_{loc}} & \widehat{\mathbf{B}^{2,2}_{loc}} & \cdots & \widehat{\mathbf{B}^{2,\beta}_{loc}} \\ \vdots & \vdots & \ddots & \vdots \\ \widehat{\mathbf{B}^{\alpha,1}_{loc}} & \widehat{\mathbf{B}^{\alpha,2}_{loc}} & \cdots & \widehat{\mathbf{B}^{\alpha,\beta}_{loc}} \end{bmatrix} \begin{bmatrix} \widehat{[\mathbf{i}^1]^n_{loc}} \\ \widehat{[\mathbf{i}^2]^n_{loc}} \\ \vdots \\ \widehat{[\mathbf{i}^\beta]^n_{loc}} \end{bmatrix} = \begin{bmatrix} \widehat{\Delta v^1_{loc}} \\ \widehat{\Delta v^2_{loc}} \\ \vdots \\ \widehat{\Delta v^\alpha_{loc}} \end{bmatrix} - \begin{bmatrix} \widehat{\Delta a^1_{loc}} \\ \widehat{\Delta a^2_{loc}} \\ \vdots \\ \widehat{\Delta a^\alpha_{loc}} \end{bmatrix} \tag{3.88}$$

After solving the wrenches, they must be converted back to full 6D wrenches and rotated back to world coordinates:

$$[\mathbf{i}^\beta]^n = \mathbf{G}^\beta_w[\mathbf{i}^\beta]^n_{loc} \tag{3.89}$$

3.10.2 Contacts & Friction

In section 3.9, our model of friction was introduced. The tangential friction will be given from equation 3.83 as part of the friction model, M_F^α. The force must be along the vector $\mathbf{n} + \mu_t \mathbf{d}_t$, hence the following constraint formulation is used:

$$(\mathbf{n} + \mu_t \mathbf{d}_t)^T [\mathbf{f}^{\alpha,i}]_n = 0 \tag{3.90}$$

This fits in our generic definition of constraints as an orthonormal basis must be specified. The orthogonal basis $(\mathbf{t}_1, \mathbf{t}_2, \mathbf{n}+\mu_t \mathbf{d}_t)$ is used to specify the linear constraint directions, where $\mathbf{t}_1$ and $\mathbf{t}_2$ are arbitrary, non-constrained, tangential directions. Notice that additionally, the normal contact force should be opposite to the normal:

$$\mathbf{n}^T [\mathbf{f}^{\alpha,i}]_n \leq 0 \tag{3.91}$$

We choose to formulate this constraint slightly different, but equivalent in practice, as:

$$(\mathbf{n} + \mu_t \mathbf{d}_t)^T [\mathbf{f}^{\alpha,i}]_n \leq 0 \tag{3.92}$$

Then the mathematical formulation of equality and inequality is consistent. In case $e_a = 1$, the angular friction must be considered as well. In this case, we have the direction of friction given directly as $\mathbf{d}_a$. The constraint is:

$$\mu_a \mathbf{d}_a [\mathbf{t}^{\alpha,i}]_n - (\mathbf{n} + \mu_t \mathbf{d}_t)^T [\mathbf{f}^{\alpha,i}]_n = 0 \qquad \text{if} \qquad e_a = 1 \tag{3.93}$$

Normally, the angular friction should only be dependent on the normal force. To avoid imposing too many constraints, we have chosen that the angular friction shall depend on the force direction given by the tangential friction and the normal force. By this definition of tangential and angular friction, the equality constraints fit into our existing definition of generic constraints. Another effect of this definition is that the resulting dynamics can be solved with a convex optimisation.

3.10.3 Solution by Iterative Optimisation

The problem is stated as an optimisation problem, similarly to the solvers previously proposed:

$$\min \frac{1}{2} \mathbf{f}^T \mathbf{f} \qquad \begin{matrix} \mathbf{A}^e \mathbf{f} \simeq \mathbf{b}^e \\ \mathbf{A}^i \mathbf{f} \leq \mathbf{b}^i \end{matrix} \qquad \mathbf{f}_i \leq \mathbf{0} \tag{3.94}$$

with N_e bilateral and N_i unilateral velocity-level constraints. Notice that this is similar to equation 3.23, except that here we have added inequality constraints. When solving for impulses, we needed to solve for equality constraints, as the restitution model gave the desired outgoing velocity directly. In this case, we must allow for contacts to leave. Therefore the inequality constraint is used. The system is solved with the LinearImpulseSolver

previously shown in algorithm 3. Now, we use $\mathbf{A} = \begin{bmatrix}\mathbf{A}^e \\ \mathbf{A}^i\end{bmatrix}$, $\mathbf{b} = \begin{bmatrix}\mathbf{b}^e \\ \mathbf{b}^i\end{bmatrix}$, $\mathbf{f} = \begin{bmatrix}\mathbf{f}^e \\ \mathbf{f}^i\end{bmatrix}$. A slightly different objective function than the one in 3.37 is used. The values of $\boldsymbol{\Delta}\mathbf{s}_B$ and $\boldsymbol{\Delta}\mathbf{s}_X$ are chosen as minus the gradient $\mathbf{g}(\mathbf{s}_B, \mathbf{s}_X)$ to the objective:

$$\begin{aligned} f(\mathbf{s}_B, \mathbf{s}_X) = \frac{1}{2}\sum_{i=1}^{N_e}[(\mathbf{s}_B)_i]^2 + \frac{1}{2}\sum_{i=N_e+1}^{N_e+N_i}\Big[&\max[(\mathbf{s}_B)_i, 0]^2 + \max[(\mathbf{s}_X)_i, 0]^2 \\ &+ \alpha\left(\max[-(\mathbf{s}_B)_i, 0]^2 + \max[-(\mathbf{s}_X)_i, 0]^2\right)\Big] \end{aligned} \tag{3.95}$$

The first sum ensures that the equality constraints are satisfied. The second sum are used for the inequality constraints. Here, the first two terms are supposed to ensure the inequality constraints, whereas the last two terms seek to minimise contact forces and velocities in the outgoing direction. The value of α should be rather small. We choose $\alpha = 0.01$, and when we are close to the optimum we set α to zero to ensure that the constraints are satisfied.

3.11 Simulated Sensors

To make simulation useful in development of robotic systems, feedback from simulations are required. Examples of sensors that can be simulated are cameras, scanners, point cloud sensors and tactile sensors. The sensors that a dynamics engine must be aware of are the latter category of tactile sensors. From a dynamic simulation point of view, the other sensors can be updated using only kinematic information about the positions of all objects in the scene.

In RobWorkSim, a distinction is made between three different types of tactile sensors: force/torque, tactile array and body contact sensors. The force/torque sensor measures the force in a constraint, E^α, as $\frac{\mathbf{f}_n^\alpha + \mathbf{f}_{n+1}^\alpha}{2}$ and $\frac{\mathbf{t}_n^\alpha + \mathbf{t}_{n+1}^\alpha}{2}$. A tactile array creates the equivalent of an array of contact constraints. If force is applied to a normal, the corresponding cell is activated. Finally, the body contact sensor measures all contact forces acting on a body. Notice that the body contact sensor is the least realistic form of sensor, hence it should be used with care.

The simulated sensors in a dynamic simulation are often quite simple as they only read out values, which are used inside the engine anyway. Note, however, that a sensor like a tactile array introduces redundancy, and if forces do not get distributed correctly, the quality of the resulting measurements will be poor.

3.12 Contact Detection & Tracking

Contact detection is an important prerequisite for realistic dynamic simulation. Ideally, contact detection is a generic component, which can be used with many different physics engines, as proposed by [41]. In practice, however, an engine will often have certain requirements to the generation of contacts that depends on the methods in use. In previous sections the new engine was introduced with all its methods. It is already clear from these methods that they rely on sophisticated contact algorithms. In fact most of the complexity in our proposed method stems from the requirements they put on the contact detection. To mention a few important requirements:

- The rollback algorithm in section 3.7 requires a continuous measure for the distance between two objects to be able to resolve the time of impact. The first contact between two bodies can easily be resolved, but if there are existing contacts, and a new contact is found, this quickly turns into a complex problem. The contacts must be tracked locally to avoid penetrations. Furthermore, a classification of the contacts must also be used to know if the contact was a new or old contact.
- In section 3.9, a friction model is presented that relies on stateful contacts due to the micro-slip modelling. Again this requires tracking of contacts.
- A classification of new and old contacts must be used to determine which new contacts to be treated as collisions in the next step of the simulation loop.

Hence, there are two important concepts. One is contact tracking, and the other is contact classification. In our simulator these two concepts are integrated into the contact detector and not into the engine itself. The engine relies on the ability to attach meta-data to each contact, which the contact detector must then track.

There are two conflicting views that the engine has been designed for:

1. It makes sense to look into idealised cases to gain basic knowledge about optimum strategies for assembly operations. For instance, a peg-in-tube scenario can be modelled with primitives and then contact detection can be performed analytically. This will be done fast and more accurate than using triangle mesh approximations. For very tight fits, mesh approximations are infeasible.
2. In general, the simulator must support triangle mesh geometries, as real world industrial objects will not be primitives.

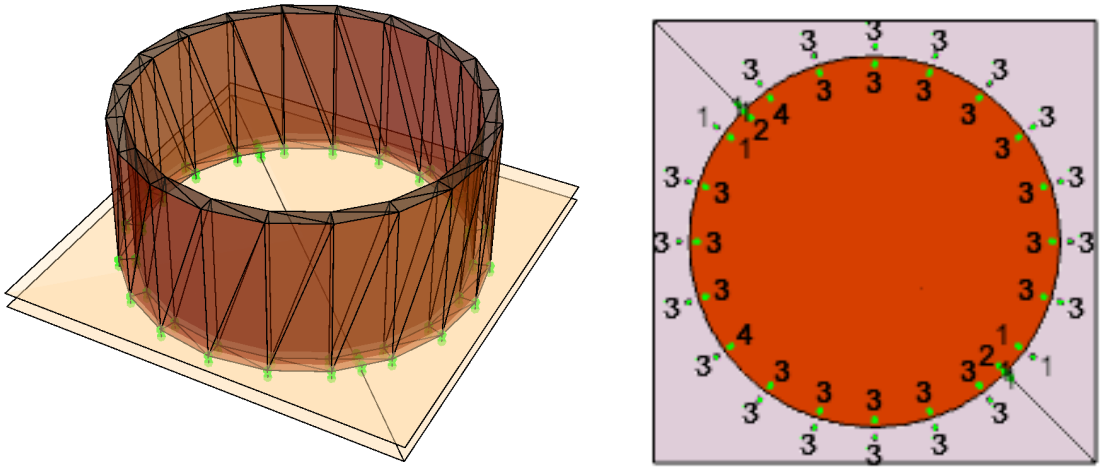
The first view has the advantage that a minimum of contacts will be returned, and tracking and classification becomes easier. Contact and depth can often be determined even if there are penetrations, meaning that a contact layer can be avoided completely. The latter is very

generic, but penetration depths and contacts are difficult to determine if a contact layer is not used. The contact layer is a problem in itself in a very tight-fitting scenario. In practice triangle models are so common that they should be supported. This requires a mapping of contacts, across contact detections, that unfortunately will require some additional threshold for classification of old and new contacts. The proposed contact handling is developed to support both of these views, though it adds a lot to the complexity.

Often, continuous contact detection is used to avoid tunnelling, where a time-step is so big that objects fly through each other without noticing. Tunnelling is not considered in this case, and we leave it up to the user to choose a suitable time-step. Continuous collision detection usually handles only linear motion without considering rotational effects, meaning that the detection is not a guarantee against tunnelling. Even though as large time-steps as possible should be used, the steps should still be small enough to make sense. It is expected that in simulation of manipulation tasks, tunneling is not a major concern, compared to penetration avoidance for instance.

For generic contact generation using triangle mesh representations, we will rely on the Proximity Query Package (PQP), which is open source. PQP uses a hierarchical representation, which uses OBBTrees (Oriented Bounding Boxes) [86]. PQP can be used for overlap testing (collision detection), computing distances and testing, if two objects are closer to each other than a specified tolerance. To be able to use PQP for contact generation, a modified version of the distance computation function is created. We will refer to this modification of PQP as RobWork-PQP. The modified distance function is called DistanceMultiThreshold. Where the purpose of the ordinary distance function is to find the closest distance between two objects and exit as soon as this distance is found, the modified method continues and finds all distances that are within some threshold. This requires more work, but will give distances for all triangle pairs inside a contact layer, making it suitable as a basis for stable contact detection. The information from PQP will involve the point pairs and their distances. In figure 3.6, an example is shown for a tube in contact with a plane. The plane is composed of two triangles, while the tube is composed of 160 triangles. Note that for a case like this, where the object is concave, the OBB tree will not be expected to be efficient as most triangle pairs are in collision anyways. As shown, many contact points are returned multiple times due to the same contacts being generated from neighbouring triangle pairs. A contact is generated from each point pair returned from PQP. Note that the PQP distance function does not work if there are penetrations, hence it is assumed that the contact pairs will always be separated by a small distance. The normal is then easily determined as the vector between the point pair, and the distance is given directly by PQP.

The number of contacts are clearly larger than required. Therefore, a filtering is performed to reduce the number of contacts. The filtering is based on the average contact point and normal for each contact pair. As a first step, contacts are grouped into clusters with similar normals (within 10°). Afterwards, a Oriented Bounding Rectangle (OBR) is fitted to each



(a) Tube to plane contacts with resolution 20

(b) 122 Contacts in Top View (46 unique)

Figure 3.6: Using a modified PQP method for contact generation between a plane and a tube. Detection requires 314 bounding volume tests and 136 triangle tests.

cluster. The OBR will try to cover as large an area as possible while keeping the deepest penetrating point. The duplicate points will automatically be handled in the filtering process. Tracking of the contacts are done by using a distance threshold. If a contact is within a certain distance from the last contact detection, it is considered the same point.

3.13 Simulation Loop

In section 3.2, the overall design of the simulator was introduced. As each part of the engine has now been described in detail, the overall simulation loop is presented in algorithm 10, which is very related to figure 3.1.

In algorithm 10, the controllers are updated as the first thing in line 2. As the controllers can change the velocity of kinematic bodies directly, the controller update must happen before collisions are handled in line 4. Besides changing the velocities of kinematic bodies, the controllers can apply forces and torques directly to bodies and constraints. Finally, constraint velocities can be set, which makes it possible to model joint motors. After handling collisions, the applied force is found using these new velocities. This could for instance be the force of a damped spring. In lines 6 to 12 the integration scheme is set, dependent on whether collisions occurred or not. In line 13, the positions are updated using rollback if required. The persistent and new contacts are manipulated according to their velocity in lines 17 and 18. In line 19, the contact force is determined using the *Contact & Constraint Solver*. Velocities are integrated in lines 20 to 23. Contacts that have too large distance are removed in line 24, and in the end, correction is done and sensors are updated. With this very brief summary, we conclude the presentation of our physics engine.

Algorithm 10 The simulation loop

function STEP($t_n, \Delta t, \mathbf{q}_n, \dot{\mathbf{q}}_n, \mathbf{f}_n, G_n^S$)
  $\begin{bmatrix} \dot{\mathbf{q}}_n^i \quad \forall \quad V^i \in \mathbb{V}_K \\ \mathbf{f}_n^{ext} \\ [\mathbf{f}_{app}^{\beta,i}]^n \\ \Delta \mathbf{v}_{n+1}^{\beta} \end{bmatrix} \leftarrow$ UPDATECONTROLLERS(t_n,Δt,$\mathbf{q}_n$) ▷ See section 3.3
  Save state in case of rollback repetition
  $\begin{bmatrix} \dot{\mathbf{q}}_n^* \\ G_n^s \end{bmatrix} \leftarrow$ COLLISIONSOLVER($\mathbf{q}_n$,$\dot{\mathbf{q}}_n$,G_n^s) ▷ Section 3.5, algorithm 1
  Calculate $\mathbf{f}_{app}^{\beta}(\mathbf{q}_n, \dot{\mathbf{q}}_n^*) \quad \forall \quad \beta$
  if $\dot{\mathbf{q}}_n^* \neq \dot{\mathbf{q}}_n$ or $t_n = 0$ **then**
    $\mathbf{f}_n^{\beta} \leftarrow \mathbf{0}$
    CLEARCONTACTSTATE($\tilde{\mathbb{E}}_i(G_n^S)$)
    INT $\leftarrow$ EULER
  else
    INT $\leftarrow$ HEUN
  end if
  $\begin{bmatrix} s \\ \mathbf{q}_{n+1}^* \\ \dot{\mathbf{q}}_{n+1}^P \end{bmatrix} \leftarrow$ POSITIONUPDATEWITHROLLBACK(Δt,INT,$\mathbf{q}_n$,$\dot{\mathbf{q}}_n^*$,$\mathbf{T}_n$) ▷ See algorithm 9
  if $s = 0$ **then**
    repeat step with augmented $\mathbb{E}_i^{new}$
  end if
  $\tilde{\mathbb{E}}_n \leftarrow \tilde{\mathbb{E}}_n \cup$ GETSMALLVELOCITY($\mathbb{E}_{n+1}^{new}$,$\dot{\mathbf{q}}_{n+1}^P$,v_{col}^{α})
  $\mathbb{E}_{n+1}^{new} \leftarrow \mathbb{E}_{n+1}^{new} \cup$ GETLARGEVELOCITY($\tilde{\mathbb{E}}_n$,$\dot{\mathbf{q}}_{n+1}^P$,v_{col}^{α})
  $\mathbf{f}_{n+1} \leftarrow$FINDFORCE($\Delta t$,INT,$\mathbf{q}_n$, $\mathbf{q}_{n+1}^*$, $\dot{\mathbf{q}}_n^*$, $\dot{\mathbf{q}}_{n+1}^P$) ▷ Section 3.10
  for all bodies $i \in \mathbb{V}_D$ **do** ▷ Can be done in parallel
    Calculate $\mathbf{f}_n^i$ and $\mathbf{f}_{n+1}^i$ ▷ Section 3.6
    $\dot{\mathbf{q}}_{n+1}^i \leftarrow$ INTEGRATEVELOCITIES(Δt,$\mathbf{g}$,INT,$\mathbb{E}_n^i$,$\mathbf{q}_n^i$, $\mathbf{q}_{n+1}^*$, $\dot{\mathbf{q}}_n^*$, $\dot{\mathbf{q}}_{n+1}^P$,$\mathbf{f}_n^i$,$\mathbf{f}_{n+1}^i$)
  end for
  REMOVECONTACTSBREAKINGAWAY ▷ With large distance
  repeat
    $\mathbf{q}_{n+1} \leftarrow$ CORRECTPOSITION($\mathbf{q}_{n+1}^*$) ▷ Section 3.8
    $\begin{bmatrix} \tilde{\mathbb{E}}_{n+1} \\ \mathbb{E}_{n+1}^{new} \\ \mathbb{T} \end{bmatrix} \leftarrow$ TRACKANDFINDCONTACTS($\mathbf{q}_{n+1}$,$\mathbb{T}$)
  until 5 iterations or $|\mathbb{E}_{n+1}^{new}| = 0$
  SENSORUPDATE(t_n,Δt,$\mathbf{q}_{n+1}$,$\dot{\mathbf{q}}_{n+1}$,$\mathbf{f}_n$,$\mathbf{f}_{n+1}$) ▷ Section 3.11
end function

CHAPTER 4

Choice of Parameters

Usually, physic engines have a wide range of parameters, which can be tuned. This can be the step size, thresholds, iteration limits, precision goals and much more. Some parameters should be chosen by the user, as they are very simulation specific, while others are more generic and should only in very advanced cases be modified by the user. In this chapter, we will try to give a hands-on discussion of how to choose parameters along with a discussion of the benefits and drawbacks of increasing or decreasing these parameters. This chapter is a good starting point for users, who need to do assembly simulation, as it will go through some of the most important parameters for this use case.

4.1 Springs & Compliance

One of the key features of RWPE is higher-order integration of springs. This is important when modelling passive compliance in a system. The task of selecting good spring parameters will be the topic of this section. The most simple kind of spring system, one can imagine, is a dynamic body connected to a fixed or kinematic body. The engine requires specification of a compliance and damping matrix. For a spring, that acts as both a linear and angular spring, the engine allows specification of a full 6×6 matrix for both. Hence, it requires 72 parameters to specify a spring. Fortunately, the user will most often be satisfied with modelling of the compliance and damping with diagonal matrices, where the linear and angular components are independent, and each principal direction is independent from the others. This reduces the number of parameters to 12. We will now try to establish some simple guidelines for the choice of these parameters, which will be related to the choice of the simulation time-step, Δt.

4.1.1 Simple Springs in One Dimension

Consider a simple linear spring in one dimension with a fixed body in one end and a dynamic body in the other end. The only forces acting on the dynamic body are gravity and the spring force:

$$f_s = -kx - cv \tag{4.1}$$

For a desired displacement, l, under gravity, g, the compliance, c, must be:

$$c = \frac{l}{mg} \tag{4.2}$$

This equation should be the basis for determining the compliance for the spring. If the compliance becomes small, the system will be stiff, and the spring might cause instability if the time step is chosen to be small. A rule of thumb, for the maximum time step to choose for a given compliance, can be determined based on the oscillation frequency for an undamped spring:

$$f = \frac{1}{2\pi\sqrt{cm}} \tag{4.3}$$

During one period, there should be at least $N = 2$ time-steps to capture the motion of the spring in the motion integrator. We base this on the Nyquist frequency. For a spring frequency f, the simulator must sample with a frequency of at least $f_{step} \geq 2f$. Hence, our stepping frequency should be:

$$f_{step} = \frac{N}{2\pi\sqrt{cm}} = \frac{1}{\Delta t} \tag{4.4}$$

where Δt is the step size. Now, the upper limit of the step size can be expressed as:

$$\Delta t_{max} = \frac{2\pi}{N}\sqrt{cm} = \frac{2\pi}{N}\sqrt{\frac{l}{g}} \tag{4.5}$$

where it is advised that the user, under normal circumstances, chooses $N > 2$. For an angular spring similar limits can be set up, but here the moment at the spring due to gravity must be used. This makes the angular compliance a bit more tricky. Let the distance from the center of mass to the spring be r, and the desired rotation due to gravity be a. The angular compliance should then be:

$$c_{ang} = \frac{a}{rmg} \tag{4.6}$$

The upper limit for the step size is in this case:

$$\Delta t_{max} = \frac{2\pi}{N}\sqrt{c_{ang}I} = \frac{2\pi}{N}\sqrt{\frac{aI}{rmg}} \tag{4.7}$$

where I is the inertia. This expression is a bit more complex and shows that the time step must be smaller if the spring acts far from the centre of mass or if the inertia is small. It is very important to consider this for long, thin objects.

After the compliance parameters have been chosen, the damping, d, can be determined as:

$$d = 2\zeta\sqrt{\frac{m}{c}} \qquad d_{ang} = 2\zeta\sqrt{\frac{I}{c_{ang}}} \tag{4.8}$$

where $\zeta = 1$ for critically damped system, $\zeta > 1$ for overdamped system and $\zeta < 1$ for underdamped system. In most circumstances, one will simply choose $\zeta = 1$.

4.1.2 More Complex Springs

Usually, we are interested in simulation of more than one-dimensional springs. For a full 6D spring, we can choose to consider each of the 6 spring dimensions independently. The compliance is then selected as the desired deflection due to a certain force or torque for

each of these directions. Notice that angular directions with small inertia must have a very compliance to avoid instabilities. We will consider the case shown in figure 1.1 as an example. The cylinder is $l_{cyl} = 10$ cm in length, the radius is $r_{cyl} = 2.5$ cm and the mass is $m = 1.54$ kg. The spring should deflect $l = 1$ mm due to gravity. Now, the compliance can be determined as:

$$c_{lin} = \frac{l}{m \cdot 9.82} = 6.6 \cdot 10^{-4} \tag{4.9}$$

In this case, we choose all linear directions to have the same compliance. The maximum time-step possible is given by:

$$\Delta t_{max} = \frac{2\pi}{2}\sqrt{c_{lin} \cdot m} = 31.6 \cdot 10^{-3} \tag{4.10}$$

For stable integration of an undamped linear spring, when only influenced by gravity, the time-step should be chosen below 30 ms as a rule of thumb.

Notice that in general, the behaviour of a spring depends on the used integration scheme. In practice, we find that $N = 8$ often seem to work well. The main purpose is to give a rough estimate of a good time-step in a generic setting. In practice, it will often be possible to choose a higher time-step if damping is used. The value for Δt_{max} should be considered an initial guess, which the user can tune according to the observed behaviour.

4.2 Contact, Rollback & Correction Layers

We will consider the issue of determining the size of contact layers. Consider the peg-in-tube case using the PQP method for detection of contacts. The cylinder and tube are both approximated by meshes with a resolution of $N = 20$. The cylinder has a radius $r_{cyl} = 2.4$ cm, and the tube has a radius of $r_{tube} = 2.5$ cm. Hence, there is a gap of 1 mm between the two bodies. Clearly, the contact and rollback layers must fulfil $\epsilon_{layer} + \epsilon_{rollback} \leq 1$ mm. Notice, however, that due to the geometric model, the minimum effective radius of the tube becomes:

$$r_{tube}^{min} = \frac{1}{\sqrt{2}}\sqrt{1 + \cos\frac{2\pi}{N}}\, r_{tube} = 2.469 \text{ cm} \tag{4.11}$$

This limits the layers to $\epsilon_{layer} + \epsilon_{rollback} \leq 0.7$ mm. Choosing a layer with this exact size might cause the entire inner surface of the tube to come into contact with the cylinder. However, this requires that they are aligned correctly. If the layers are bigger than 0.7 mm, the correction algorithm will have a hard time determining a correction, as any correction will still cause the contacts to penetrate the contact layer. To have some safety and room for the rollback layer, it is proposed to set the contact layer to $\epsilon_{layer} = 0.5$ mm or less for this particular case. Notice that especially for concave objects, like the tube, one must be careful determining the size of the contact layer. Also, consider the case where the fit is tighter and has a gap of only 0.1 mm (this is not unrealistic). In this case, N must be at least 36, almost doubling the number of triangles that need to be handled in the contact detector. If there should be room for a contact layer as well, we must have $N > 36$. In this case, the mesh

representation is clearly not good. Instead, the objects should be represented by primitives directly, requiring dedicated methods for contact detection.

If a contact layer is chosen, $\epsilon_{layer} = 0.5$ mm, we must be careful that the rollback threshold is set appropriately relative to the contact layer. In this case, $\epsilon_{rollback} = 0.1$ mm might be appropriate. It is important that $\epsilon_{rollback} < \epsilon_{layer}$ to avoid excessive penetration of the layer at initial colliding contacts. This can work in some cases, but a smaller rollback layer gives a higher safety margin for avoidance of hard collisions, in which the entire contact layer is penetrated. Notice, however, that a very small rollback layer might cause more time spent on resolving the exact time of impact for new collisions.

Consider the correction layer for inequalities, $\epsilon^{in}_{correction}$. The correction algorithm will allow some contacts to be violated slightly within this layer. See section 3.8. Penetration of the contact layer should never be greater than $\epsilon^{in}_{correction}$. For the PQP method, it is obvious that $\epsilon^{in}_{correction} < \epsilon_{layer}$. If a contact penetrates the layer more than the size of ϵ_{layer}, it is impossible to recover as bodies are now in collision (which can not be handled for mesh geometries). Similar to the rollback layer, it is advised that the correction layer is kept significantly lower than the contact layer. The correction layer should, however, still provide a bit of room, giving the correction solver a bit more freedom.

Finally, we will briefly consider the same parameters in the case of contact detection strategies that handle penetrations. In this case, we choose $\epsilon_{layer} = 0$, as no contact layer is required. If there is a gap of 1 mm, the rollback and correction layers should be set smaller than this. For instance the rollback layer could be set to 0.1 mm. If the contacts should be completely hard, the correction layer should be set to zero. To give the correction solver a bit more flexibility, the layer can be set to a small value.

4.3 Collisions & Time of Impact

By default, collisions are handled with the hybrid collision solver presented in section 3.5.2. We will look into the parameters related to the resolution of the collisions, but first it is important to discuss at what time we will consider a new contact a collision. A new colliding contact must have a relative velocity component, in the penetrating direction, greater than v^{α}_{col} to be considered a collision. By default, this velocity is $v^{\alpha}_{col} = 1\frac{\text{cm}}{\text{s}}$. If a new contact does not have this velocity, it is simply treated as a persistent contact without handling collisions first. Notice that as the time-step, Δt, gets smaller, large forces can be required to correct such an penetrating velocity. It is advised that the threshold, v^{α}_{col}, is decreased in this case. If the value is chosen too big, no collisions will be handled with rollback and restitution models. This will cause large interaction force if the time-step is small. If on the other hand v^{α}_{col} is set to zero, all new contacts will be handled with impulses. Notice, however, that new contacts, which do not have much ingoing velocity, can occur. If this is the case, the rollback method will most likely fail to find a time of impact, and much time will be spend resolving collisions

that are not there. In summary, choose the threshold for penetrating collision velocity large enough, that only real collisions are handled with impulses, and low enough that this does not result in excessive contact force if handled as a persistent contact. Next, consider the resolution of the time of impact. This is done using a rollback algorithm. This algorithm will return a time of impact when the colliding contact is within a layer, $\epsilon_{rollback}$. The default value of this threshold is fairly small for most applications, namely 0.01 mm. If the exact time of impact is less important, increase this value to allow the time of impact to be found faster. If set very high, the collisions are not applied at the time of impact, and penetrations will occur. This will step the simulation forward faster as the time-step does not have to be decreased if collisions occur. Making the layer small will take too much time to resolve the exact time of impact, which might not be important. On the other hand, this will make sure that penetrations are avoided.

Following the discussion on when collisions occur, we will now consider the parameters related to handling of collisions in the simultaneous and hybrid collision solvers, presented in sections 3.5.1 and 3.5.2. *The hybrid solver* specifies velocity thresholds for constraints, $\epsilon_{\Delta v}$ and $\epsilon_{\Delta w}$, and contacts, $\epsilon_{contact}$. See algorithm 5. These thresholds determine how long impulses will propagate. If set too small, impulses can potentially propagate back and forth for a long time until the velocity constraints are satisfied. If set too high, the solver will not respect the post-condition that it must satisfy: that no collisions may occur after collision solving. Notice that this post-condition is difficult to satisfy in practice for constraints that must have zero relative velocity after collision solving. In practice, this will never hold true, so instead a tolerance is used. Notice that the thresholds used in the hybrid solver do not have anything to do with v_{col}^{α}, which was discussed before. Where v_{col}^{α} determines when the collision solver is invoked, the three thresholds in the hybrid solver control when collisions have been handled sufficiently. At this point, notice that only contacts can initiate collisions, but at the propagation stage collisions occur in both contacts and constraints. The default values for the three thresholds are as follows:

$$\epsilon_{\Delta v} = 10^{-8}, \qquad \epsilon_{\Delta w} = 10^{-8}, \qquad \epsilon_{contact} = 10^{-4}, \qquad i_{max} = 1000 \tag{4.12}$$

In practice these thresholds for constraints are very small and can in many cases be increased to reduce time spent in the hybrid solver. A maximum of 1000 iterations are done. For small systems in manipulation, the method will, however, be expected to find a solution fairly fast.

In *the simultaneous solver*, there are a lot of parameters that can be tweaked. The most important ones are the ones related to the objective function. The numerical method for solution of impulses was presented as algorithm 4. The set of default parameters is as follows:

$$\begin{aligned} \epsilon_{SVD}^{rel} &= 10^{-6} & e &= 10^{-6} & k_{max} &= 1000 \\ \gamma_{max} &= 100 & \alpha &= 0.01 & \alpha_t &= 10^{-5} \end{aligned} \tag{4.13}$$

The relative SVD precision was defined in equation 3.24. Typically, there will not be a need to modify this value. The target error, e, determines the stop criteria for the iterative algorithm.

The error term is related to the change in the resulting impulse vector. If the mass of bodies in the system is small, e might have to be decreased (because small impulses are expected). If the mass of bodies is big, e might have to be increased. This is an unfortunate dependency on the size of the impulses in the system. In the future we will work on making this threshold dimensionless. The γ value determines the importance of satisfying equality constraints, over the objective that the impulse vector has a certain direction for inequality constraints. The purpose is to bias the solver towards a quick solution of the equality constraints. As these are satisfied, the γ is automatically lowered by the engine (see equation 3.38), favouring the objectives on the impulses. Under normal circumstances, it should not be necessary to adjust this parameter significantly. The α is a term in the objective function that tries to minimise the found impulses. This is mainly used in redundant configurations to obtain a distributed impulse. When we get close to the solution, the α parameter is set to zero. The α_t threshold controls this. In general, α_t should be larger than e. Many of these parameters are quite specific, so unless the user has very specific needs, the default ones will mostly be okay. Notice, however, that e might need to be adjusted according to the size of the impulses that are expected in the system.

4.4 Contact & Constraint Solver

For the contact and constraint solver, many parameters are similar to the ones discussed in section 4.3. For the constraint solver we use the following default parameters:

$$\begin{aligned} \epsilon_{SVD}^{rel} &= 10^{-6} & e &= 10^{-3} & k_{max} &= 200 \\ \gamma_{max} &= 2500 & \alpha &= 0.01 & \alpha_t &= 10^{-2} \end{aligned} \tag{4.14}$$

Notice again that e and α_t are not dimensionless entities. If large forces are expected in the system, these values can be increased. If small forces are expected the values can be decreased. This is a trade-off between the number of iterations in the solver and the amount of constraint violation. The α parameter controls the objective, which will cause distributed loads. See also equation 3.95. Increasing the value will cause the solver to prioritise small forces rather than satisfaction of constraints. Making it zero will lose the distributed load objective completely. This might be acceptable for some simulations.

We have now considered some of the most important parameters of the engine, with the hope that the user can understand the effect of changing each individual parameter. In practice it is however difficult to see the effect of changing the parameters, and hence it is difficult to optimise them to the specific needs in a given application. In chapter 6 different software initiatives will be presented that will facilitate visualisation of the effect of choosing different parameters, which we hope will make this process more user-friendly.

CHAPTER 5
Qualitative Tests & Evaluation

To illustrate some of the concepts in the developed engine, different qualitative tests will be performed. The tests are partly inspired by other works on engine evaluation [41, 63, 79, 80]. Tests performed in this chapter include integration of unconstrained motion, collision and friction modelling, compliance and redundancy. The purpose of the tests is not to stress test the engines, but to test different basic aspects of simulation with the simplest tests.

5.1 Test Setup

First of all, the generic test setup for all tests in this chapter must be considered. All engines make it possible to set many different parameters and options. Some of these options are generic and some are very specific to a particular simulation. In this section, the options that are generic for all the performed tests will be discussed. First of all, the engines have been compiled individually using options that make the engines comparable. Table 5.1 gives an overview of the engine versions and the compile options used.

<table>
<tr><th></th><th>Open Dynamics Engine</th><th>Bullet Physics</th><th>RobWorkPhysicsEngine</th></tr>
<tr><td>Version</td><td>0.13</td><td>2.83.5</td><td>1.0</td></tr>
<tr><td>Release Date</td><td>4 February 2014</td><td>5 June 2015</td><td></td></tr>
<tr><td rowspan="2">Options</td><td>Disabled Asserts</td><td colspan="2">Release Mode</td></tr>
<tr><td colspan="3">Double Precision
Shared Libraries</td></tr>
</table>

Table 5.1: Versions and build-options used.

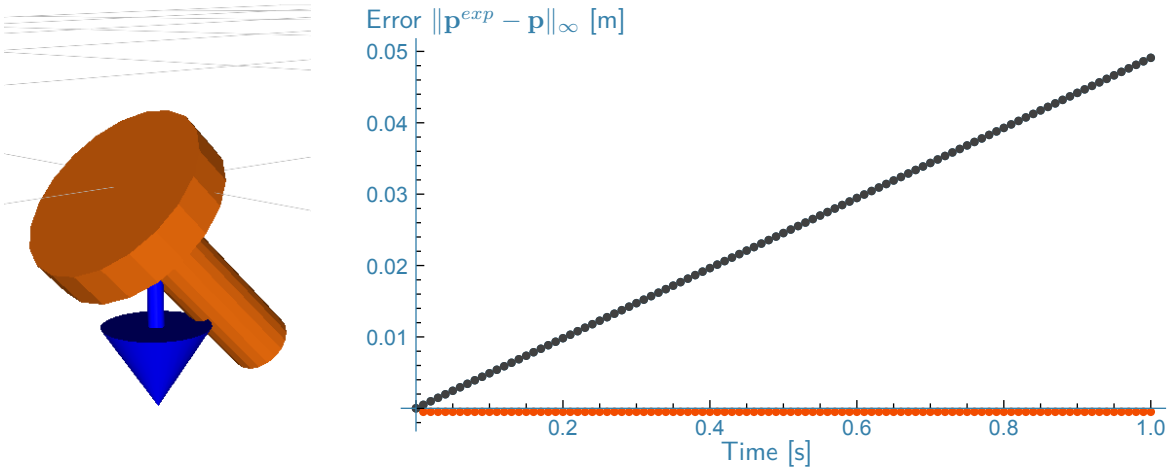
All tests are performed on Ubuntu 14.04.3. Note that the packages libbulletdynamics2.81 and libode1 (version 0.11.1) are available from the standard repositories. These versions are from 2012 and 2009 respectively and have not been used as they are too outdated.

5.2 Integrator Tests

Three experiments are performed with the main purpose of testing the motion integrators, which are used in common engines. There will be only one object in the scene thus no contacts or constraints will be possible. This makes the tests very simple as no collisions can occur and no inverse dynamics will need to be performed.

5.2.1 Linear Motion

Dropping an object under gravity, $\mathbf{g}$, cause the object to follow the ideal trajectory, $\mathbf{p}^{exp} = \frac{1}{2}\mathbf{g}t^2$. Figure 5.1 shows the deviation from this ideal trajectory using different engines.



(a) Free-falling object. (b) Resulting error from expected trajectory. ● RWPE ● ODE & Bullet

Figure 5.1: Positional error for a free-falling object under gravity $\|\mathbf{g}\| = -9.82$. Time-step is $\Delta t = 0.01$.

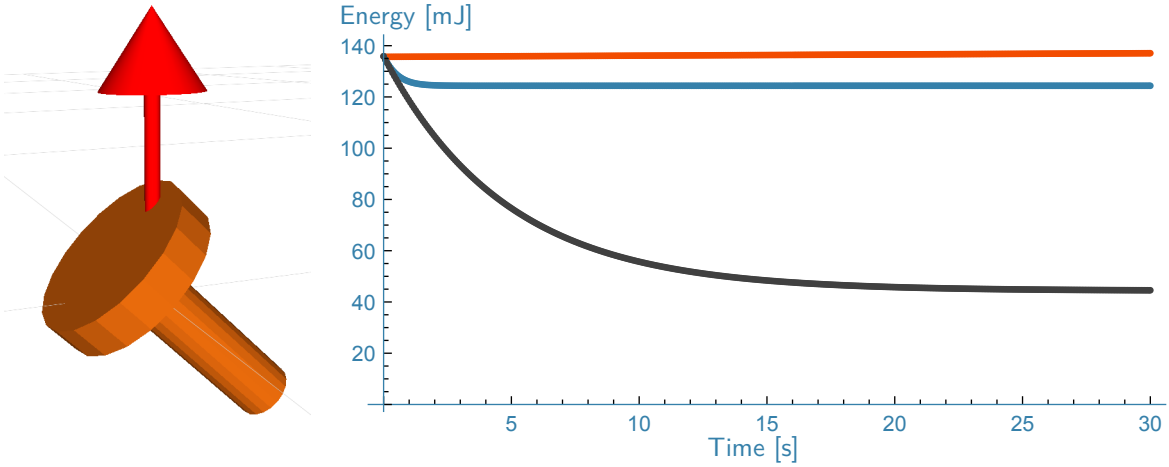
As ODE & Bullet use the semi-implicit Euler integration scheme, the position will be integrated in each time-step using a velocity that is slightly too large. This causes the positional error to increase over time as the object drops a bit faster than it should. Opposite to this, the Heun method does not deviate from the expected trajectory. The second-order nature of this integration method is thus able to integrate ordinary gravity correctly. Note that all simulators give the correct velocity, $\mathbf{v} = \mathbf{g}t$. Hence, it is only the position that deviates from the expected trajectory. Note also that the trajectory obtained with the RWPE engine is in fact $\mathbf{p} = \frac{1}{2}\mathbf{g}(t^2 - \Delta t^2)$. This is due to the first step where the semi-implicit Euler scheme is used before switching to the Heun scheme. The positional error will be $p^{exp}(1) = -4.91$, meaning that the error is roughly 1%.

Often, an object will only be free-falling in a very short period during simulation as it will come to rest due to contacts with other objects. However, it is quite interesting that the most used engines in robotics research can not even simulate a free-falling object correctly.

In [80] a similar test was performed. Here, ODE, OpenTissue and TrueAxis gave the same results as expected from the semi-implicit Euler method. Bullet and Tokamak gave a different result, which is similar to a so-called second-order Euler. However, it is unclear exactly what this refers to. Clearly, we get same results for both ODE and Bullet in this test, indicating that there must have been changes in either ODE or Bullet since then.

5.2.2 Angular Motion

Consider the free-floating object shown in figure 5.2a. The object will have an initial angular velocity in the direction shown in the figure, and the object will not be subject to gravity. The inertia around the center of mass is $I_{\parallel} = 0.0011243$ along the symmetry axis, and $I_{\perp} = 0.0023164$ around the orthogonal axes. Note that the angular velocity does not coincide with the principal axes of inertia, and the simulation is expected to show a torque-free precession phenomena occurring. The Bullet engine had the gyroscopic force disabled by default prior to version 2.83, so previous versions will give different results. Figure 5.2b



(a) Rotating object. (b) Kinetic energy profile. ● RWPE ● ODE ● Bullet

Figure 5.2: Energy conservation under torque-free precession. The initial angular velocity is 2 revolutions per second in the direction shown with the red arrow. The angle between the principal axis of inertia and the angular velocity is 45 degrees. The timestep used is $\Delta t = 0.01$.

shows the results of this test, which is best illustrated by a graph of the kinetic energy during the simulation. Under ideal circumstances, the energy will be conserved throughout the simulation.

ODE integrates the gyroscopic force using the semi-implicit Euler scheme. It can be seen that implicit integration of the gyroscopic force with the semi-implicit Euler scheme loses some energy, but it reaches a stable plateau after a short period of time. Overall, the first order semi-implicit Euler method performs well for this type of motion as it gives a stable integration. On the other hand, it causes the gyroscopic term to vanish very quickly due to the angular velocity vector that aligns with the principal inertia direction with the maximum inertia. Note that stable integration of gyroscopic forces is a feature of ODE version 0.13. Prior to this, a similar test will show a gain in energy causing instability.

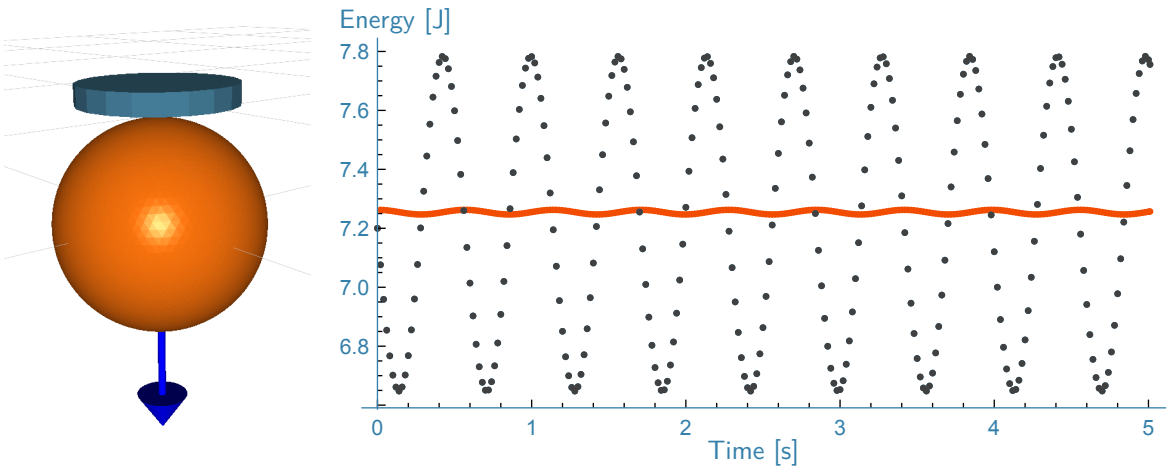
Bullet allows three different modes of handling of the gyroscopic force: explicit, implicit in body frame and implicit in world frame. The default is implicit in body frame, and this is the one used in figure 5.2b. A significant drop in energy is seen. In this case the angular

velocity vector aligns with a minimum principal inertia direction. If the gyroscopic force is integrated implicitly in the world frame instead, we see a energy that is preserved even better than RWPE.

The RWPE engine shows almost perfect preservation of the energy during the 20 seconds of simulation time. This is due to the second-order Heun integration scheme, which does a better job at capturing the gyroscopic term in the motion equations. Note that simulation over a longer time will cause the Heun method to gain energy. After 10 minutes the energy gain will be close to 0.12 mJ, which is negligible when compared to the other methods. Using the Heun integration scheme will allow use to use larger timesteps, while still being able to simulate rotating objects realistically.

5.2.3 Spring Integration

A spring test is performed to test the motion integration using a different kind of external force. Figure 5.3a shows a ball that is attached to a fixed body by a virtual spring. The spring is undamped so the energy should be preserved and the ball should continue to move up and down indefinitely. Figure 5.3b shows the energy profile of the system during simulation. As



(a) Ball attached to spring

(b) Energy profile. ● RWPE ● ODE & Bullet (coincident)

Figure 5.3: Integration of undamped spring for different engines. The timestep used is $\Delta t = 0.01$. The mass of the ball is 32.9 kg, the spring force is $f_s = -1000\Delta x$ with initially extended spring of 120 mm.

can be seen, the *ODE & Bullet* engines have the same behaviour. This is expected as they use the same integration scheme. The energy has large fluctuations, but energy is conserved in average. This is a great property of the semi-implicit Euler scheme. However, note that *RWPE* has a more steady energy profile. There is still fluctuations, but the higher-order integration causes the energy to be more stable.

For reference, figure 5.4 shows a theoretical result of a spring simulation using different first order integration schemes. As can be seen, the Explicit euler scheme can be expected to

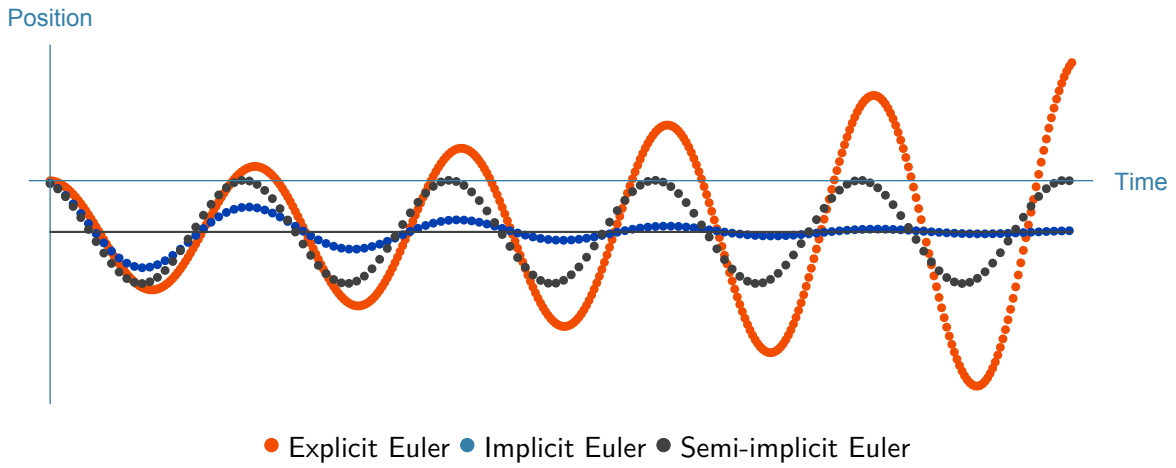


Figure 5.4: Theoretical comparison of first order integrators.

be unstable for integration of a spring, while the implicit scheme is highly stable. Unfortunately, the implicit spring is damped by the integration scheme. Damping is often desired in simulation as it makes the simulation more stable. However, in our case it is important that the spring is simulated correctly according to specification. Note that the correct behaviour is captured by the semi-implicit integration even though the energy levels can fluctuate during simulation.

5.3 Collision Solver & Restitution

A few tests are performed with focus on the collision aspects to test that collisions are simulated realistically according to the specified model of restitution. The tests performed in the following sections will have increasing complexity, from a single collision between one object pair to multiple collisions with many objects, which require impulse propagation.

5.3.1 Simple Bouncing Ball

The most basic collision test to perform is a ball bouncing on a plane. The ball is released from a specific height and accelerates due to gravity. Then it bounces on the ground a few times before settling. The restitution model determines how much energy is removed at each collision. A low value will cause the ball to settle quickly, while a high value will make it bounce for a longer period of time. The position of the ball is plotted in figure 5.5 for three different engines dropping the ball from a height of 30 cm and using a restitution of 0.75. The solid line shows an analytic reference solution. For contact detection, ODE and Bullet use their own internal sphere to plane contact detection algorithms. The RWPE engine uses a sphere to plane algorithm with tracking that allows rollback.

The ODE engine is a fixed time-step engine which means that the ball and plane can penetrate at collisions. This can be mitigated through the use of a larger contact layer or a smaller

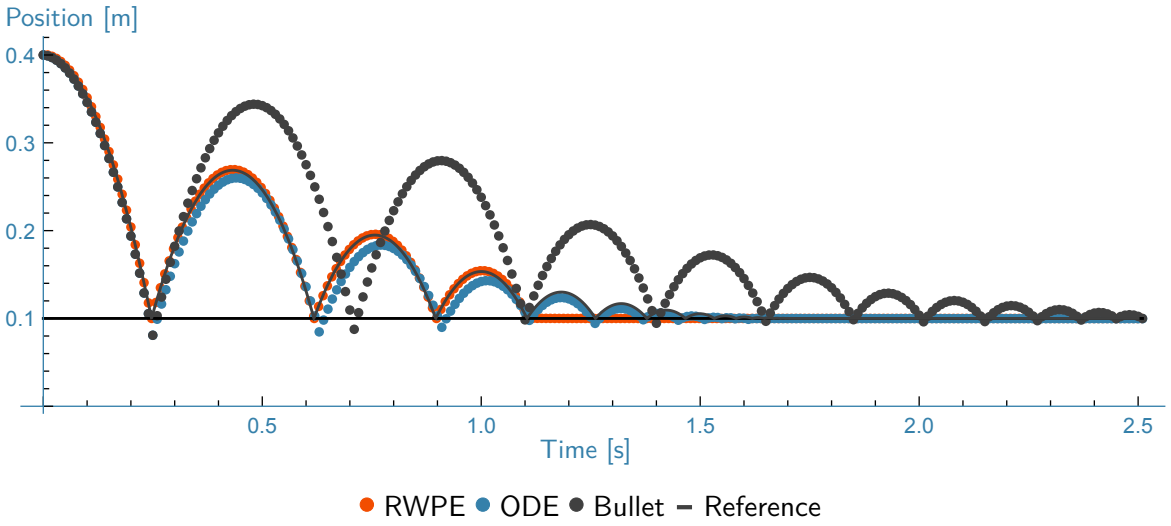


Figure 5.5: Position of a bouncing ball with radius 10 cm when released with the center of mass at 40 cm above ground. Gravity is $\|\mathbf{g}\| = -9.82$ and restitution is 0.75. Time-step is $\Delta t = 0.025$ s.

step-size. The penetration will cause a wrong time of impact, and the trajectory will quickly deviate from the ideal. However, it does follow the ideal trajectory very closely.

The Bullet engine also runs in fixed time-step mode. With Bullet it seems that the restitution is actually bigger than what was requested. It turns out that running Bullet with an ERP value of zero will cause the same bouncing behaviour as in ODE. Hence, the restitution in Bullet is actually dependent on the error reduction value. This makes the resitution very difficult to control.

The RWPE engine clearly captures the expected motion and comes closest to the ideal reference trajectory. This is both due to adaptive time-stepping and a better integration scheme. A lower order integration scheme is used for one simulation step at each collision. This might explain the slight deviation from the ideal trajectory over time. Please note that the RWPE engine only has 4 collisions before settling. This is due to thresholds set in the simulation. We refer to section 4.3 for a discussion of the relevant parameters.

In [80], a similar test was performed, but here a ball was dropped on top of a resting box. It was found that Bullet and Novodex came closest to the expected bouncing height followed by Newton and Tokamak. It is argued that it is not important with the correct bouncing height, whereas it is more important that there is some connection between an increase in restitution and the bouncing height. We do not agree with this for scientific use even though it is true for games. Also, it is uncertain what the expected outcome is when collisions with multiple bodies occur. For this reason the expected bouncing height seems a bit unjustified in the previous comparison study.

5.3.2 Bouncing Cylinder

A single ball bouncing on a plane is the simplest collision possible. Therefore, a cylinder is tested to increase complexity. The cylinder is tilted and dropped on a fixed plane. The first collision with the plane will cause the cylinder to gain a large angular velocity. Quickly hereafter a new collision will happen at the other end, and so forth. The important thing to consider is if the cylinder is actually able to come to rest, or if the cylinder will continue to jiggle. The latter will typically be possible in a sequential impulse-based method. The

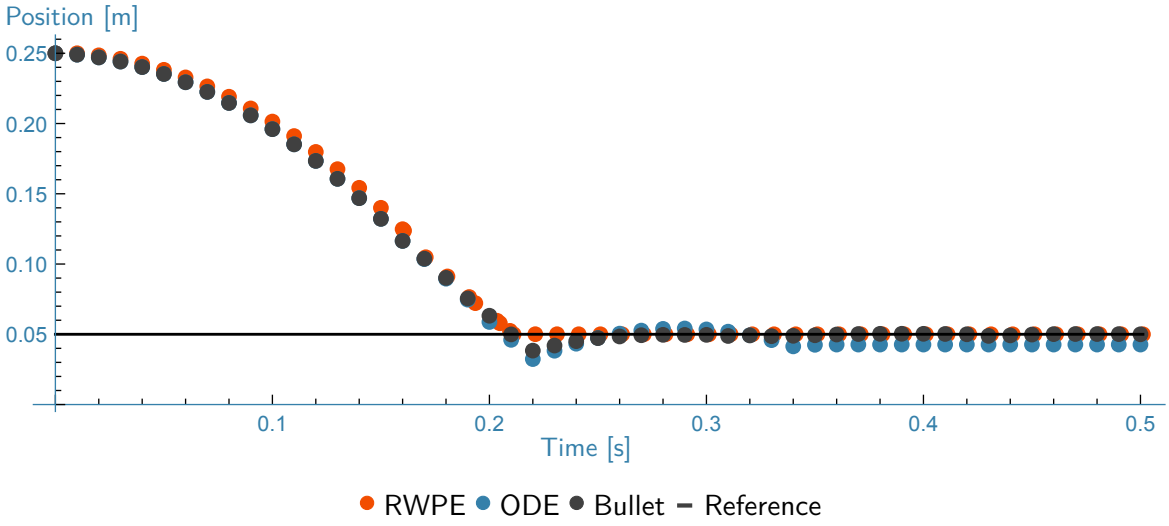


Figure 5.6: The trajectory of the centre of a cylinder dropped onto a plane. Restitution coefficent is $\xi = 0.25$, length of the cylinder is 25 cm and radius is 5 cm. Initial tilt angle is 45° and simulation is done using the time-step $\Delta t = 0.01$.

trajectory of the centre of mass for the cylinder is shown in figure 5.6 for the three engines. As the radius of the cylinder is 5 cm, it should never be possible for the centre of the cylinder to get below the 5 cm reference line. However, notice that both ODE and Bullet cause some severe penetrations of the layer. ODE comes to rest with a slight penetration, while Bullet seems to correct completely.

The corresponding angular velocities are shown in figure 5.7. Both ODE and RWPE come to rest, while Bullet starts rotating with a quite large velocity. This is a rotation around the centre line of the cylinder. It is uncertain exactly why this effect occurs. Our test shows that the rollback method resolved the time of impact such that penetrations are avoided completely. Also, the body comes to rest without any penetration.

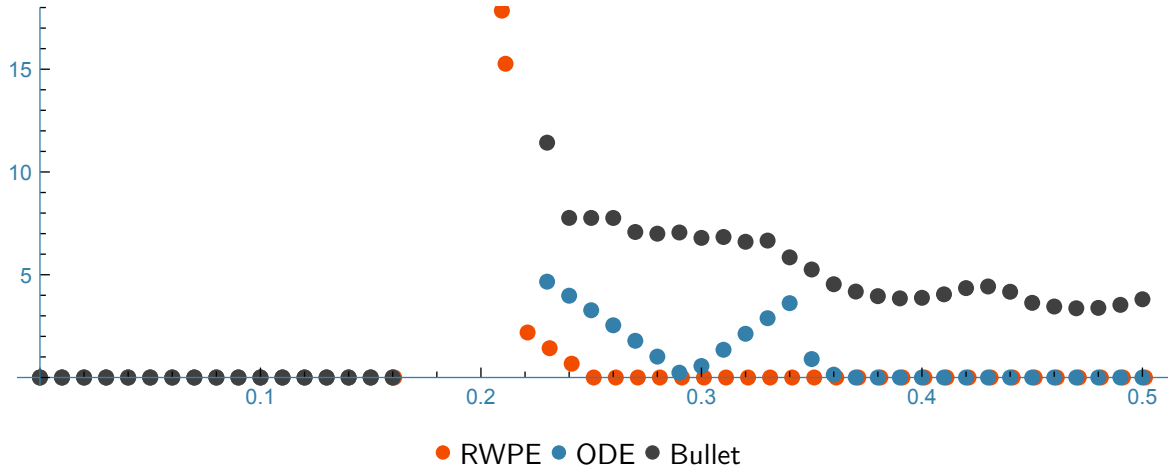


Figure 5.7: Angular velocity of cylinder when dropped onto a plane

5.4 Static Friction

A test is performed where a box lies on an inclined plane. The box is expected to start sliding if there is no friction. With increasing friction, the box will be able to resist sliding for increasingly steep inclines. Figure 5.8 shows the result of a test that illustrates this concept. For reference, the expected angle where the box starts sliding is $\tan^{-1}\mu$, which is shown in

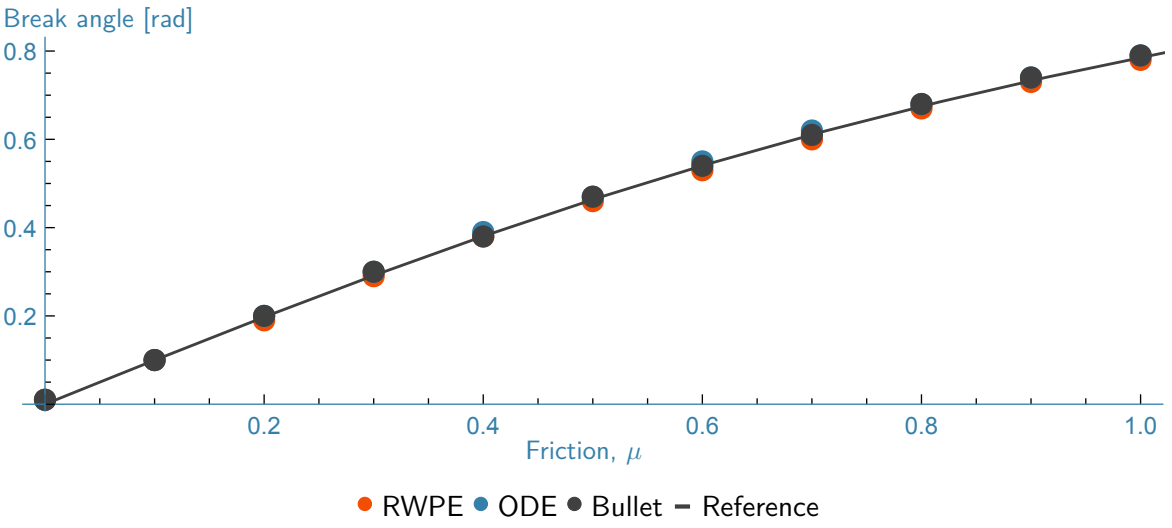


Figure 5.8: The incline of a plane where a box will start sliding.

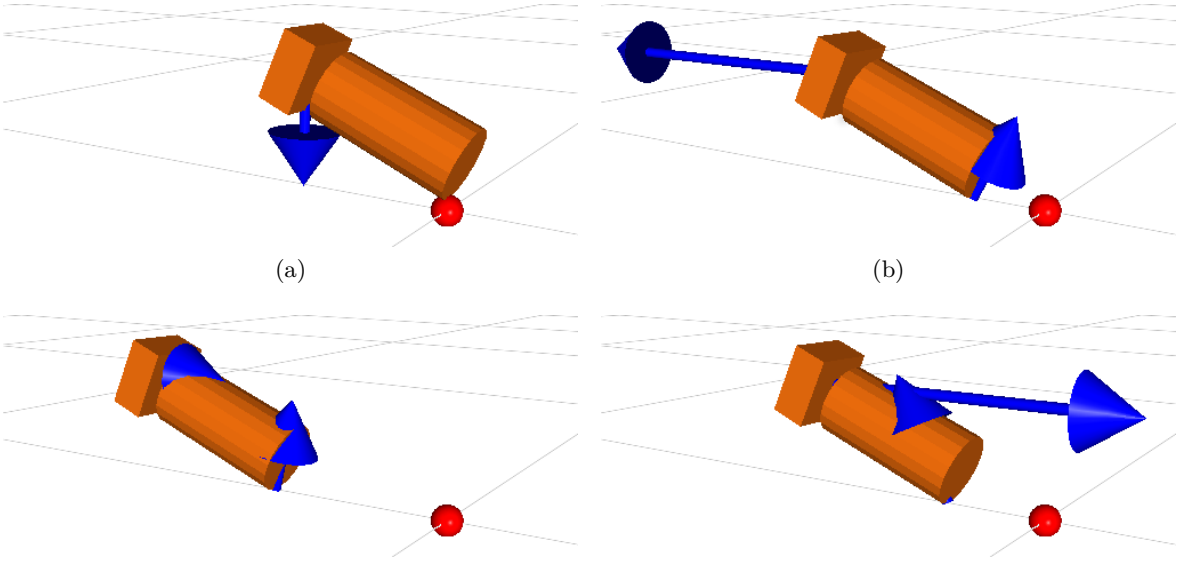
the figure. In the RWPE, the angle where motion starts is close to what is expected compared to ODE and Bullet. Note that in the RWPE engine, the box will initially slide a bit before the micro-slip model kicks in.

In [80] a static friction test was performed similarly to the one performed here. ODE was not tested, but Bullet was shown to start sliding at a much lower angle than it should to obey the static friction set by the user. Luckily, it has not been possible to confirm this as all

engines have shown good correlation between the friction coefficient and the angle of sliding in our test.

5.5 Compliant Motion with Friction & Restitution

Compliance and friction will now be tested using the motion shown in figure 5.9. The cylinder is attached to a kinematic box by a compliant spring. Initially, the box is moved downward as in figure 5.9a. The cylinder will then make contact with the ground, and the spring will make sure that there is a steady normal force. After time $t > 0.25$ the velocity of the box is set as a sinusoidal function with a maximum velocity of $0.2\frac{m}{s}$ and a frequency of 0.5 Hz. In figure 5.9a the box starts moving in the positive direction. The cylinder will maintain the normal force in the contact with the ground due to the spring. However, the relative motion will cause friction between the cylinder and ground. This friction is opposite to the direction of motion as shown in figure 5.9b. When velocity decreases and begins to transition into the opposite direction of motion, figure 5.9c shows the micro-friction behaviour. In this case the friction and sliding direction is actually in the same direction for a short while. Finally, in figure 5.9d the speed and friction are again opposite. In our test, the effective friction



(a) (b) (c) (d)

Figure 5.9: The scene used to test compliance and friction modelling.

coefficient has been determined as the relation between normal and tangential contact force. This effective friction is shown as a function of the relative velocity in figure 5.10. Notice that we clearly see the micro-slip model working for the RWPE. Bullet uses the Coulomb friction model exactly as expected. However, ODE does show a slightly different behaviour than expected. It seems that the friction coefficient becomes different for each direction of motion. Recall the discussion about ODE and friction in section 2.3. ODE computes the normal force assuming no friction. The maximum friction force is then found from the normal force and

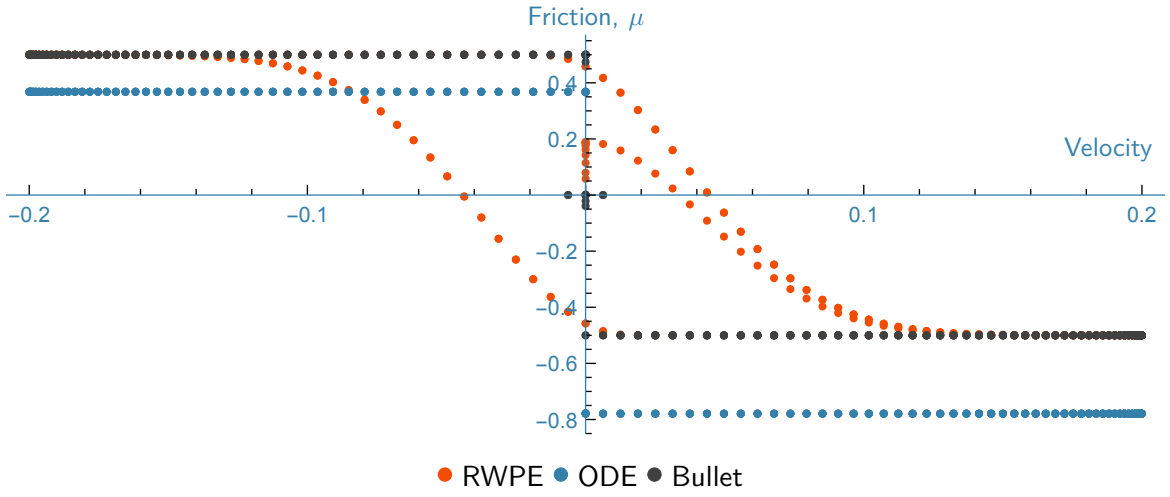


Figure 5.10: The effective friction coefficient for sinusoidal relative velocity. Reference friction coefficient is $\mu_C = 0.5$. The micro-slip model with Coulomb friction is used in RWPE.

the coefficient of friction. Now, ODE solves the LCP problem using this maximum friction for both tangential directions independently. This causes the applied tangential friction force to be the same for motion in both directions. It makes the friction rather un-physical, which is also acknowledged by ODE [91].

5.6 Redundant Configurations & Balanced Forces

The scene shown in figure 5.11 has been used to test how the solver handles redundant configurations. The box is controlled kinematically, and the cylinder is connected to the box by a spring. The tube is in resting contact with the floor, and the kinematically controlled box is moved downwards until the cylinder touches the tube as shown in the figure.

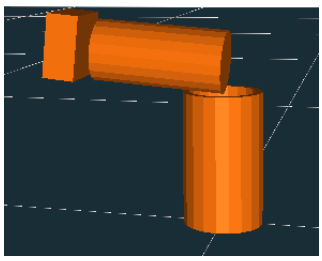


Figure 5.11: The scene used for testing the distribution of forces in the RWPE engine.

Figure 5.12 shows the contact forces measured between the cylinder and tube. RWPE gives a nice and steady contact force while ODE fluctuates. This can be due to unstable contacts with the floor and the fact that the contacts are redundant. The force measured in the RWPE is preferable compared to ODE for a control strategy used to assemble the cylinder and tube. Notice that in Bullet, the contact impulses are extracted after a simulation step

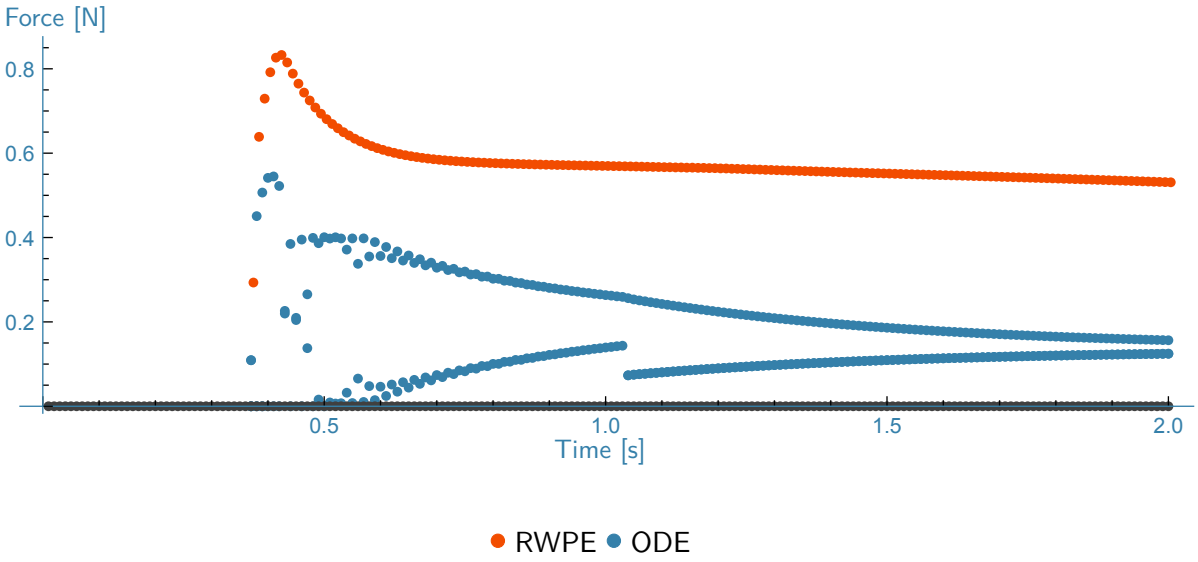


Figure 5.12: Contact force measured between the compliant cylinder and the tube.

using the GETAPPLIEDIMPULSE method on contacts. This is basically an impulse that we try to convert to a force by dividing by the size of the time-step, Δt. It seems that extraction of such information does not work. This may be due to impulses being cleared during collision resolution.

The minimum distance between the tube and floor is shown in figure 5.13. Here it is seen

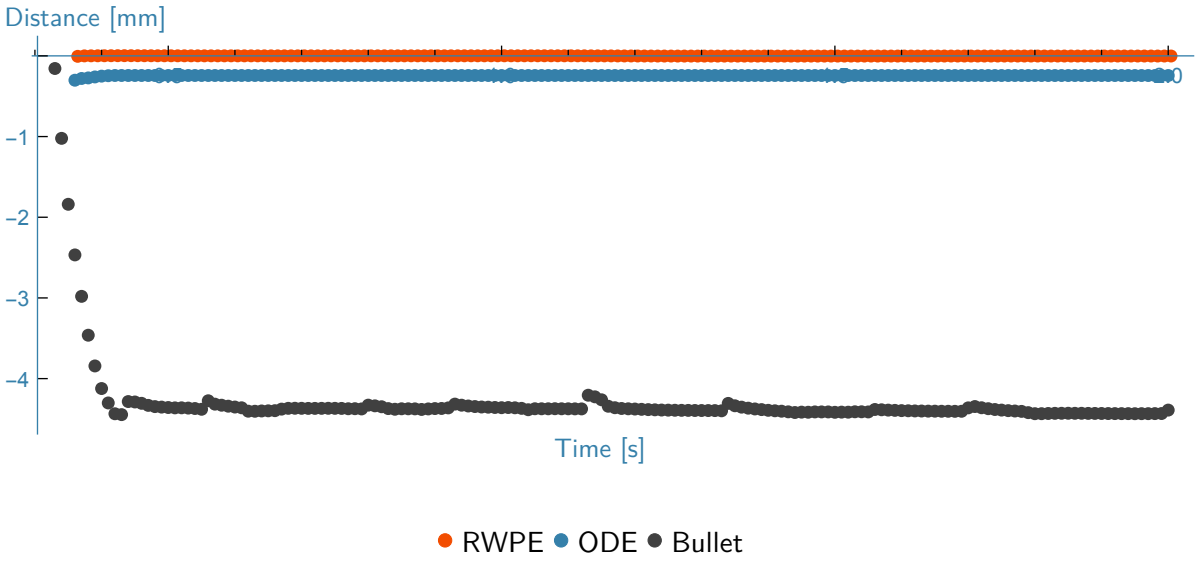


Figure 5.13: The minimum distance between the compliant cylinder and the tube.

that ODE penetrates due to the soft contacts. This will also explain that the contact force in figure 5.13 is smaller than the corresponding contact force measured in RWPE. The spring will not deflect as much, due to the penetration, and therefore the spring does not apply the same amount of force. Finally, the Bullet penetration is clearly very large. This is expected

to be due to the fact that Bullet is an impulse-based solver that does sequential handling of collisions. Notice that we use the internal spring of generic 6D constraint in Bullet.

The performance of the engines has been measured in figure 5.14. It is clearly seen that

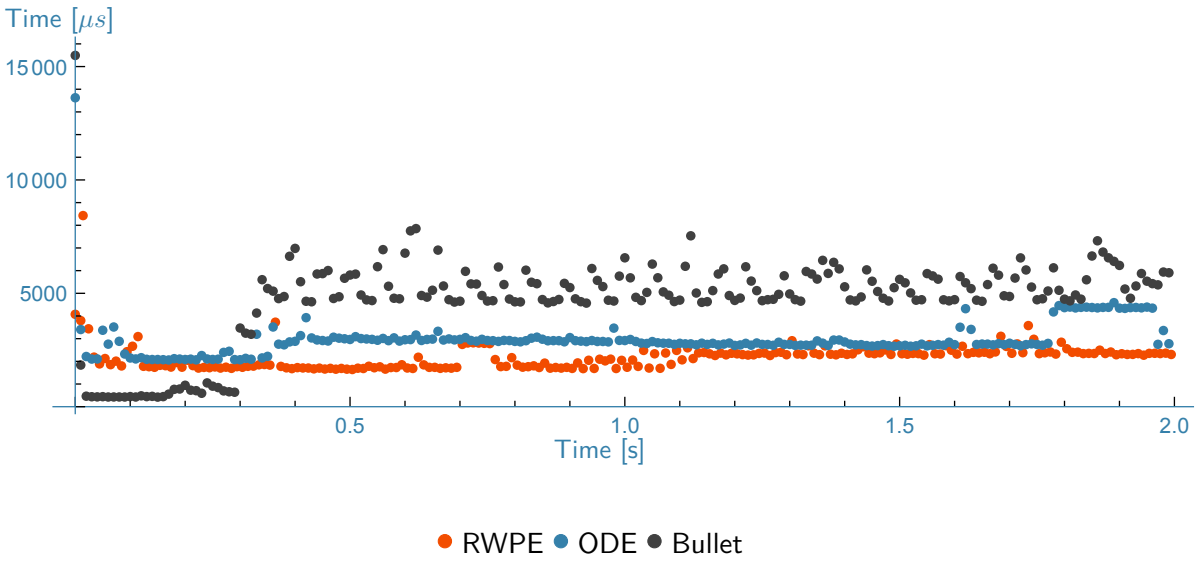


Figure 5.14: The computation time for a simulation with a compliant cylinder resting on a tube.

Bullet uses most time followed by ODE, and finally RWPE uses least time. Note that initially both ODE and RWPE take longer time before the cylinder touches the tube. This is probably due to the redundancy and Bullet using a lot of time to apply impulses sequentially until constraints are satisfied. In this case, the constraint-based solvers deal with the redundancy directly, while RWPE is the only one being able to do so without any penetrations.

The qualitative tests of the three engines show many differences regarding physical accuracy. Some of the tests in this chapter are fairly simple, and it is somewhat surprising that engines used in robotics research cannot even integrate simple, free motion correctly, and that they have difficulties when it comes to modelling restitution and friction phenomenas. However, notice that ODE and Bullet are engines developed with games and animation in mind. Therefore, it is not fair to say that they perform badly. They just try to solve a different problem than us.

CHAPTER 6

Software Framework

In this chapter, we will look into the overall software framework that RWPE fits into. The first part will introduce how to set up a model of the system to simulate and how to actually run a simulation. After this, we will give concrete examples of how parameters can be tuned, and how the engine can be extended with custom models and methods. In the final part, we will discuss tools that facilitate debugging, optimum usage of resources through adaptive parallelisation and a generic test suite for evaluation of engines.

6.1 The RobWork Framework

All work presented in this thesis was implemented in the RobWork framework [2]. RobWork is the core component, which provides basic mathematics, path planning algorithms, work-cell and kinematic simulation concepts. The RobWorkStudio component builds on top of RobWork and provides a convenient graphical user interface. The RobWorkSim component [39], which was mainly developed with the Open Dynamics Engine as the underlying engine, can be used for dynamic simulation. Both RobWorkStudio and RobWorkSim are designed to be independent of each other. This allows scaling simulation experiments in headless environments such as high-performance computer clusters. However, dynamic simulation can also be used by RobWorkStudio by implementing a simulation plugin for RobWorkSim. The plugin structure is an important feature of the framework that makes the framework very customisable.

The contributions to the RobWork framework are as follows:

Assembly Task Format A subsystem was added in RobWork for specification of assembly actions. This is described in detail in chapter 7.

Java/MatLab Interface A Java interface was generated using SWIG [92]. As MatLab can interface Java, RobWork methods can be executed from MatLab. The main purpose of this feature is that control algorithms can be developed in MatLab by researchers who are more familiar with MatLab. See section 6.8.

Mathematica Interface For visualisation purposes a convenient interface was added for Mathematica. This allows evaluation of Mathematica functions from RobWork and thus ease visualisation in the simulator test framework. This will be discussed in section 6.6.

Assimp Support Integration of a third party library allowing a much broader range of geometry formats [82].

Inertia Estimation Implementation of an improved method for calculating inertia for triangle mesh geometries, based on Brian Mirtich's method [84].

The contributions to RobWorkSim are:

The RobWorkPhysicsEngine As described in chapter 3.

Bullet Physics Interface Development of an interface for Bullet Physics in addition to Open Dynamics Engine and RobWorkPhysicsEngine.

A Flexible Contact Detection Framework As described in section 3.12.

Primitive Contact Detection Strategies For contact detection between cylinders and tubes for instance.

Assembly Simulator Based on the Assembly Task Format implemented in RobWork. This is described in chapter 7.

Generic Constraints & Compliance Modelling A concept for flexible constraint definition and compliance modelling was implemented in the workcell format and simulators.

Simulator Logging A framework for detailed logging of internal values in the simulator. Provides a very user-friendly interface that allows users to follow each step of the loop in 3.1.

Dynamic Test Framework A framework for definition of tests for dynamic simulators. The purpose of the framework is to unify unit tests and comparison tests as performed in 5. The framework is discussed in section 6.9.

In the following sections some of these contributions will be explained. The dynamic workcell format will first be explained, and we will give an example of how to run a simple simulation. Then more advanced usage of the simulator is discussed, including extensions and detailed logging.

6.2 The Dynamic Workcell Format

Specification of a dynamic workcell is done in two parts. First, an ordinary workcell is defined, which specifies kinematic properties of the system. The dynamic workcell is then an extension on top, that allows specification of the dynamic parameters of the system. Both formats are specified in XML. We will begin our introduction to the format with the definition of a dynamic workcell. A simple scene, which consists of a tube, a cylinder and a box, is shown in figure 6.1 an example scene is shown.

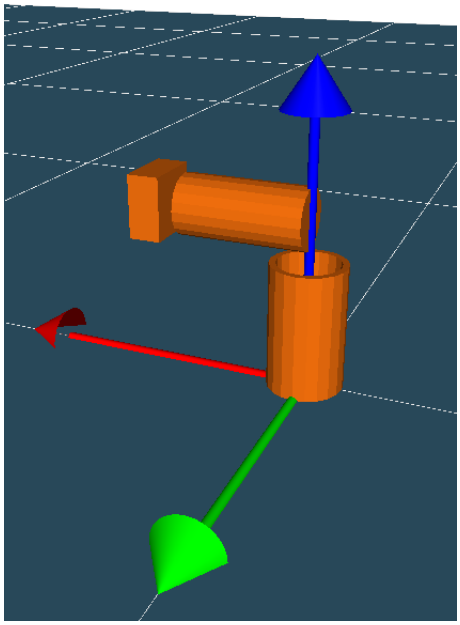


Figure 6.1: An example scene of peg-in-tube assembly. The World coordinate system is shown with x-, y- and z-directions marked in red, green and blue respectively.

```
<WorkCell name="PegInTube">
   <Frame name="Floor" refframe="WORLD">
      <Drawable name="PlaneGeo"><Plane /></Drawable>
   </Frame>
   <Frame name="Box" refframe="WORLD" type="Movable">
      <Pos>0.122 0 0.135</Pos>
      <RPY>0 -90 0</RPY>
      <Drawable name="BoxGeo">
         <Box x="0.05" y="0.05" z="0.025" />
      </Drawable>
   </Frame>
   <Frame name="Cylinder" refframe="Box" type="Movable">
      <Pos>0 0 0.022</Pos>
      <RPY>0 0 0</RPY>
      <Drawable name="CylinderGeo">
         <Pos>0 0 0.05</Pos>
         <RPY>0 0 0</RPY>
         <Cylinder radius="0.024" z="0.1" />
      </Drawable>
   </Frame>
   <Frame name="Tube" refframe="WORLD" type="Movable">
      <Pos>0 0 0.0505</Pos>
      <RPY>0 0 0</RPY>
      <Drawable name="TubeGeo">
         <Tube radius="0.025" thickness="0.005" z="0.1" />
      </Drawable>
   </Frame>
   <ProximitySetup file="ProximitySetup.xml" />
</WorkCell>
```

Listing 6.1: Example of the RobWork-XML format for specification of a kinematic WorkCell

```
<ProximitySetup UseIncludeAll="true" UseExcludeStaticPairs="true">
   <Exclude PatternA="Box" PatternB="*"/>
   <Exclude PatternA="Floor" PatternB="Cylinder"/>
</ProximitySetup>
```

Listing 6.2: Example of a ProximitySetup for a WorkCell, that allows excluding certain geometry pairs from collision detection.

The RobWork-XML format for the example scene in figure 6.1, is shown in listing 6.1. The outermost WorkCell tag defines the WorkCell and assigns it a name. The name helps RobWorkStudio to distinguish whether a scene is reloaded or if a new scene is loaded. Four Frame tags are defined for the floor, box, cylinder and tube respectively. The Floor frame is defined relative to the WORLD frame, which is a basic reference frame that always exists in a WorkCell. In line 3 a Drawable tag is added to associate geometry with the Floor frame. The geometry is named to distinguish multiple geometries on a frame from each other. The Plane tag defines a simple plane geometry. Now, for the Box frame given in lines 5 to 11, the type is set to Movable. This tag means that the frames' relative transformation to its reference frame is allowed to change. The Floor frame on the other hand is by default a frame of type Fixed, meaning it can never change. Hence, this frame definition does not require any dynamic state to be stored to specify its position. The initial position of the movable box frame is set in line 6 and 7 and gives the relative transform from its parent in the form of a position and a RPY specification of the relative orientation. Again, a Drawable is attached to the frame, and in line 9 a box is defined with the given dimensions. This gives the location and orientation of a box as shown in figure 6.1. The Cylinder frame is similar to the previous frame definitions with the difference that it is defined relative to the Box frame in line 12. This means that it will move when the box frame is moved. The definition of a Cylinder primitive geometry is shown in line 18, and this time it is defined with a transform given in line 16 and 17. This transform positions the cylinder relative to the associated Cylinder frame. Finally, the Tube frame is defined in line 21-27, with the tube definition given in line 25. In line 28 a ProximitySetup file is given by filename. The ProximitySetup makes it possible to exclude certain geometry pairs from collision and contact detection, as shown in listing 6.2. In line 1, the base settings is given. UseIncludeAll specifies that by default all geometry pairs are used in collision detection, and UseExcludeStaticPairs specifies that all fixed geometry pairs are excluded automatically. In this case, only the floor geometry is fixed and will not have any effect. As there are four geometries in the scene that can move relative to each other, the default pairs that are being collision checked are: Floor-Box, Floor-Cylinder, Floor-Tube, Box-Cylinder, Box-Tube and Cylinder-Tube. Later, we will use the WorkCell as the basis of a Dynamic WorkCell definition, in which the box will be kinematically controlled, the cylinder will be attached to the box by a spring and the tube will be freely moving. If it is assumed that the kinematic box is controlled in a sensible fashion, and that the spring will not deflect enough to make the cylinder get in contact with the floor, the two exclusion patterns in line 2 and 3 can be used. Patterns match frame names using regular expressions. The remaining non-excluded collision pairs in the workcell are Floor-Tube and Cylinder-Tube. Note that in this case there are only 4 frames, and no significant speed-up is to be expected as we use collision strategies with bounding volume hierarchies. However, note that the number of geometry pairs to test for collision increases dramatically with the number of specified geometries. This will thus be important in more complex scenes.

Now the definition of a kinematic workcell is given, and it is possible to move the three movable frames in RobWorkStudio by altering their transformations relative to other arbitrary frames in the workcell. Definition of a Dynamic WorkCell is shown in listing 6.3. The DynamicWorkCell definition is given in line 1 by referring to the ordinary kinematic WorkCell. The gravity vector, $\mathbf{g}$, is set in the world reference frame in line 2. A database of material properties is given in line 3. An example of this is shown in listing 6.4, where friction and restitution parameters are defined. The four objects from before are now defined as either fixed, kinematic or dynamic bodies. In lines 4 to 6 the Floor frame is defined as a FixedBody with a material identifier. The same is done for the Box, which is specified as a KinematicBody in line 7 to 9. The cylinder is defined as a dynamic body by the RigidBody tag in line 10 to 20. A dynamic body requires specification of mass parameters with the Mass, Inertia and COG tags. Also, an integrator can be specified. This allows the user to implement custom integration schemes in the RWPE engine. The name of the integrator refers to the corresponding integrator plugin name in this case. Extensions will be discussed further in section 6.5. In line 21 to 26, a similar definition for the Tube body is given. In this case, the EstimateInertia tag is given to automatically estimate the inertia and center of gravity, using the mass and the geometric information. Finally, the spring is specified in line 27 to 50. Springs are associated to constraints and work for the free dimensions based on the constraint type. In this case, a so-called Free constraint, which simply has 6 degrees of freedom, is specified. This allows the spring to work in full 6D. The two bodies that the constraint and spring work between are specified in line 27. The transform given in line 28 to 31 specifies the location relative to the parent where the constraint and spring work. Finally, lines 32 to 48 define the spring with compliance and damping parameters as full 6D matrices. The damping is set to critically damped as given in equation 4.8.

```
<DynamicWorkcell workcell="scene.wc.xml">
   <Gravity>0 0 -9.82</Gravity>
   <Include file="DynamicMaterialDataBase.xml"/>
   <FixedBody frame="Floor">
      <MaterialID>Aluminium</MaterialID>
   </FixedBody>
   <KinematicBody frame="Cup">
      <MaterialID>Aluminium</MaterialID>
   </KinematicBody>
   <RigidBody frame="Box">
      <Mass>0.12</Mass>
      <Inertia>
         0.000103 0 0
         0 0.000103 0
         0 0 0.000006
      </Inertia>
      <COG>0 0 0.05</COG>
      <Integrator>Heun</Integrator>
      <MaterialID>Aluminium</MaterialID>
   </RigidBody>
   <RigidBody frame="Tube">
      <Mass>0.12</Mass>
      <EstimateInertia />
      <Integrator>Heun</Integrator>
      <MaterialID>Aluminium</MaterialID>
   </RigidBody>
   <Constraint name="Spring" type="Free" parent="Box" child="Cylinder">
      <Transform3D>
         <Pos>0 0 0.01</Pos>
         <RPY>0 0 0</RPY>
      </Transform3D>
      <Spring>
         <Compliance>
            0.001 0 0 0 0 0
            0 0.001 0 0 0 0
            0 0 0.001 0 0 0
            0 0 0 0.5 0 0
            0 0 0 0 0.5 0
            0 0 0 0 0 0.5
         </Compliance>
         <Damping>
            21.91 0 0 0 0 0
            0 21.91 0 0 0 0
            0 0 21.91 0 0 0
            0 0 0 0.01 0 0
            0 0 0 0 0.01 0
            0 0 0 0 0 0.01
         </Damping>
      </Spring>
   </Constraint>
</DynamicWorkcell>
```

Listing 6.3: Example of a RobWork Dynamic WorkCell definition

An example of a minimal specification of material parameters is given in listing 6.4. First, the available materials are defined in the MaterialData section in line 2 to 5. These gives the allowed MaterialIDs in the Dynamic WorkCell definition. If no MaterialID tag is given, the default material in line 3 is used for all bodies. In lines 7 to 14 the friction map is given as the friction between each pair of material. In this section, micro-slip friction is specified. The default micro-slip friction does not exhibit Stribeck behaviour, so only one friction coefficient is specified along with the γ and r values for the hysteresis. In lines 15 to 20 different object

```
1  <IncludeData>
2     <MaterialData>
3        <Default>Aluminium</Default>
4        <Material id="Aluminium" />
5     </MaterialData>
6     <FrictionMap>
7        <Pair first="Aluminium" second="Aluminium">
8           <FrictionData type="MicroSlip">
9              <Gamma>250.0</Gamma>
10             <r>1.0</r>
11             <Mu>0.1</Mu>
12          </FrictionData>
13       </Pair>
14    </FrictionMap>
15    <ObjectTypeData>
16       <Default>hardObj</Default>
17       <ObjectType id="hardObj">
18          <Description>A hard object. with low elasticity</Description>
19       </ObjectType>
20    </ObjectTypeData>
21    <ContactMap>
22       <Pair first="hardObj" second="hardObj">
23          <ContactData type="Newton"><cr>0.0</cr></ContactData>
24       </Pair>
25    </ContactMap>
26 </IncludeData>
```

Listing 6.4: Example of a minimal material map for specification of friction and restitution.

types can be defined. Object types define how an object reacts to collisions. In lines 21 to 25 a contact map is given and specifies the coefficient of restitution for the only given object type.

6.3 Running a Simulation

To run a simulation, consider the small program in listing 6.5. In line 17 the RobWork

```
1  #include <rw/RobWork.hpp>
2  #include <rw/loaders/path/PathLoader.hpp>
3  #include <rwsim/loaders/DynamicWorkCellLoader.hpp>
4  #include <rwsim/simulator/PhysicsEngine.hpp>
5  #include <rwsim/simulator/DynamicSimulator.hpp>
6
7  using namespace rw;
8  using namespace rw::common;
9  using namespace rw::loaders;
10 using namespace rw::kinematics;
11 using namespace rw::trajectory;
12 using namespace rwsim::dynamics;
13 using namespace rwsim::loaders;
14 using namespace rwsim::simulator;
15
16 int main() {
17    RobWork::init();
18    const DynamicWorkCell::Ptr dwc = DynamicWorkCellLoader::load("DynWC.dwc.xml");
19    const PhysicsEngine::Ptr engine =
          PhysicsEngine::Factory::makePhysicsEngine("RWPEIsland",dwc);
20    const DynamicSimulator::Ptr simulator = ownedPtr(new
          DynamicSimulator(dwc,engine));
21    State state = dwc->getWorkcell()->getDefaultState();
22    double time = 0;
23    simulator->init(state);
24    TimedStatePath trajectory;
25    trajectory.push_back(TimedState(time,state));
26    do {
27       simulator->step(0.01);
28       state = simulator->getState();
29       time = simulator->getTime();
30       trajectory.push_back(TimedState(time,state));
31    } while (time < 1.0);
32    simulator->exitPhysics();
33    PathLoader::storeTimedStatePath(*dwc->getWorkcell(),trajectory,"result.rwplay");
34 }
```

Listing 6.5: Program for running a simulation with logging.

framework is initialised. This will dynamically load plugins located in standard directories. This will be the topic of section 6.5. Notice that the RWPE engine is itself an extension to the PhysicsEngine interface, provided by a separate plugin library. In line 18 the dynamic workcell, discussed in section 6.2, is loaded. The engine and dynamic simulator are created in lines 19 and 20. Notice that the dynamic simulator uses a dynamic workcell and an engine to do the actual dynamics on this dynamic workcell. State and time variables are introduced in lines 21 and 22, and the simulator is initialised with the default initial state in line 23. The default state includes the initial positions and velocities, $\mathbf{q}_0$ and $\dot{\mathbf{q}}_0 = \mathbf{0}$. A timed state path

is created and it will basically store a list of time-position pairs, $(t_n, \mathbf{q}_n)$. At line 33 this path is saved to a *.rwplay* file, which can be played back in RobWorkStudio. In line 26 to 31 the simulator is run for one second using a time-step of $\Delta t = 0.01$. Notice that the engine might divide the time-step using rollback. The GETTIME function should therefore be used to get the actual time after the time-step. Finally, in line 32 the simulator and engine are exited. Notice that this small program together with the workcell definitions in section 6.2 will make it possible to run a very simple simulation where bodies are supposed to fall into a static equilibrium due to gravity. We have not considered any actuation in this example. Instead we simply present a minimum working example of execution of a dynamic simulation.

6.4 Tuning Properties

In chapter 4, the choice of suitable parameters for the RobWorkPhysicsEngine was discussed. Now, an example is given in listing 6.6 that shows how parameters can be specified in the DynamicWorkcell format. The names of all parameters will not be listed, but instead these can be found in the documentation of the API [93]. We show just a few properties to illustrate

```
1  <DynamicWorkcell workcell="scene.wc.xml">
2     ...
3     <PhysicsEngine>
4        <Property name="RWPECollisionSolver">Chain</Property>
5        <Property name="RWPERollbackMethod">Ridder</Property>
6        <Property name="RWPECorrection" type="int">1</Property>
7        <Property name="RWPERollback" type="int">1</Property>
8        <Property name="RWPECollisionSolverPropagateConstraintLinear" type="float">
9           0.0001
10       </Property>
11       <Property name="RWPESolverIterativeSVDAlpha" type="float">0.01</Property>
12    </PhysicsEngine>
13    ...
14 </DynamicWorkcell>
```

Listing 6.6: Example of Parameter Tuning

the concept. In line 4, *the chain solver* described in section 3.5.4 is chosen as the collision solver. Notice that the name can also refer to a customised collision solver provided by a plugin. This is described in section 6.5. In line 5, Ridder's method is used for rollback, as described in section 3.7. In lines 6 and 7 the correction and rollback methods can be disabled. This is of course not encouraged, but can be useful in debugging scenarios. An example of two very specific properties is given in lines 8 to 11. In lines 8 to 10 the parameter, $\epsilon_{\Delta v}$, is adjusted. This parameter was given in equation 4.12. Finally, in line 11 the α-parameter is as given in equation 4.14.

6.5 Extending the Engine

In figure 6.2 an overview of the extendable modules of the engine is shown. Modules can be implemented as plugins to the engine through the RobWork plugin structure. It is, for instance, possible to change the way collisions are handled or the method used to solve for the contact and constraint forces. Examples of less invasive extensions are custom restitution

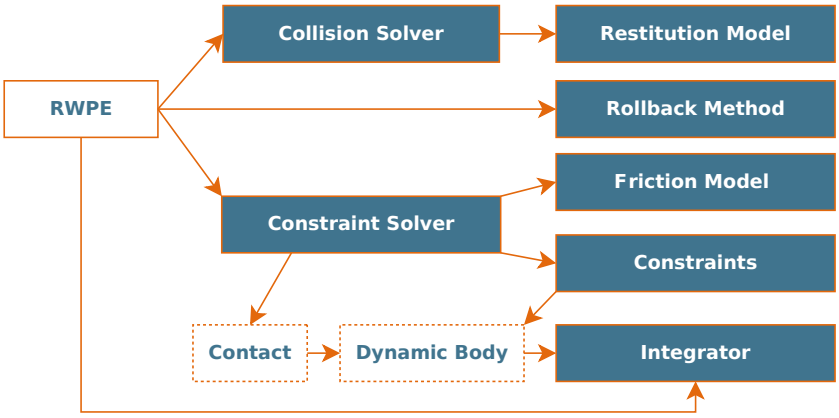


Figure 6.2: The extendable modules of the RobWorkPhysicsEngine.

and friction models. These alter the way the existing solvers handle collision and friction respectively. Furthermore, it is possible to change the SearchMethod used for rollback and to implement new integration schemes. The constraint interface is defined very generic, such that implementation of specific constraint types is possible. In the following, we will introduce the plugin structure and then consider how to extend each individual module in turn.

6.5.1 The RobWork Plugin Structure

The RobWork framework provides the ability to define so-called extension points. An extension point is determined uniquely by its identifier. Extensions can then be implemented with reference to this identifier. A registry of all extensions in the system is maintained. Typically, one or more extensions will be compiled into a separate library (a plugin), which is then loaded dynamically. The plugin will provide a list of descriptors for each extension that the plugin provides. This is meta-data, which provides information about the identifier of the extension and the extension point it attaches to. When a specific extension is requested by the user, the plugin is responsible for instantiation of the extension.

6.5.2 Example of a Friction Model Extension

As an example, consider extension of the engine with a custom friction model. The friction model was considered in 3.9, and the interface allows the user to implement a custom friction model giving the friction tuple, M_F^α, in equation 3.83. Notice that in listing 6.4 the definition of friction allowed specification of micro-slip friction with Coulomb as the gross model. Now we extend this model with a custom model including rotational friction. First, a new friction model is implemented in listing 6.7. Notice that the example is slightly simplified. The core

```
1  class CustomFriction: public MicroSlipModel {
2  public:
3     CustomFriction(const PropertyMap &map):
4        MicroSlipModel(map),
5        _muAng(map.get<double>("MuAng",0))
6     {}
7     virtual FrictionParameters getFriction(const Contact& contact,
8        const IslandState& state, const FrictionModelData* data) const
9     {
10       FrictionParameters M = MicroSlipModel::getFriction(contact, state, data);
11       if (_muAng != 0) {
12          M.enableAngular = true;
13          M.angular = _muAng;
14       }
15       return M;
16    }
17 private:
18    double _muAng;
19 };
```

Listing 6.7: Example of a custom friction model.

functionality is inherited from the default micro-slip model. In lines 3 to 6 the constructor extracts the angular friction coefficient from a PropertyMap. This is an additional value specified in the material map as shown in listing 6.8. In lines 7 to 16 the friction values, M_F^α, are created. The standard micro-slip model is used as a basis in line 10. If a friction coefficient different than zero is specified for the angular friction the angular friction is enabled in lines 12 and 13. Notice that so far, only the model has been implemented. To make the model

```
1  <FrictionData type="Custom">
2     <Gamma>250.0</Gamma>
3     <r>1.0</r>
4     <Mu>0.1</Mu>
5     <MuAng>0.1</MuAng>
6  </FrictionData>
```

Listing 6.8: Input parameters to the cusomized model.

available for the engine, it must be defined as an extension and compiled into a plugin that can be loaded by the simulator.

```
1  RW_ADD_PLUGIN(FrictionPlugin);
2  class FrictionPlugin: public Plugin {
3  public:
4     FrictionPlugin(): Plugin("FrictionPlugin", "FrictionPlugin", "1.0");
5      std::vector<Extension::Descriptor> getExtensionDescriptors() {
6        std::vector<Extension::Descriptor> exts;
7        exts.push_back( Extension::Descriptor("CustomFriction","FrictionModel"));
8        exts.back().getProperties().set<std::string>( "modelID", "CustomFriction");
9        return exts;
10     }
11     Extension::Ptr makeExtension(const std::string& id) {
12       if(id=="CustomFriction"){
13          Extension::Ptr extension = ownedPtr(new Extension("CustomFriction",
               "FrictionModel",this, ownedPtr(new CustomFriction()) ) );
14          extension->getProperties().set<std::string>("modelID", "CustomFriction");
15          return extension;
16       }
17       return NULL;
18     }
19 };
```

Listing 6.9: Example of a plugin providing a custom friction model.

The plugin is created in line 4, in which the name and version of the plugin are given. In lines 5 to 10 the descriptors are created with meta-data for the extensions provided by the plugin. In lines 7 and 8 the name of the extension and the extension point "FrictionModel" are given. In lines 11 to 18 the plugin constructs the extension when queried. Lines 13 and 14 set the meta-data similarly to before. In line 13 the friction model is instantiated.

Notice that through the plugin structure and the property map abstraction, the engine has been extended with a new, simple model of angular friction with a few lines of code. Implementation of a custom restitution model follows a similar approach. For the friction model, information is given about the contact and the state of the system. This makes it possible to make detailed models based on positions and velocities. The friction model can also store frictional state, which is tracked in the contacts during simulation. In this example, tracking of state is used for modelling of micro-slip.

6.5.3 Other Extensions

The interface for implementation of a SearchMethod for rollback is quite simple:

$$s \leftarrow \text{SearchMethod}(\mathbb{D}, \text{data})$$

It is used in the narrow-phase rollback algorithm 6. The distance set set, $\mathbb{D}$, gives the distance for each frame-pair used for rollback. This distance is given for each sampled step size so far. Currently, Ridder's method is used by default, but custom search methods can easily be implemented through the extension mechanism.

Specific constraint and spring types can be implemented through a unified interface. This is a concept explained in section 3.5.1, where the concept of a linear and angular orthonormal basis, $\mathbf{R}^{\alpha}_{s,lin}$ and $\mathbf{R}^{\alpha}_{s,ang}$, was introduced. Now, we consider how a revolute joint type can be implemented using this approach. In this case, the orthonormal basises are simply given by the identity matrix $\mathbf{R}^{\alpha}_{s,lin} = \mathbf{R}^{\alpha}_{s,ang} = \mathbf{J}$. Our selection matrix is then chosen as $\begin{bmatrix}1 & 1 & 1 & | & 1 & 1 & 0\end{bmatrix}^T$ to select velocity constraints in all linear directions and in the angular x- and y-directions. Hence, only the angular z-direction can move freely. Contrary to an engine like Open Dynamics Engine with a limited set of joint types, the generic constraint type allows for a wide array of joint definitions. Notice that a constraint is specified relative to the parent body. A constraint can store both an applied wrench and a constraint wrench as determined by the constraint solver. The extendable constraint type definition also allows implementation of custom applied wrench models, which are different than the standard damped spring model. In figure 6.2, a dependency is shown from the constraint definition to the integration scheme via dynamic bodies. In our framework, a constraint must provide the rotated and diminished version of the linear relationship shown in equation 3.86. This gives the relative velocity of the constrained directions as a linear combination of the other constraints in the system. The constraint velocity is given directly for a kinematic body, but for a dynamic body the integrator for the specific dynamic body must provide the velocity given in equation 3.85. Normally, a custom implementation of a constraint does not need to alter this default behaviour.

Finally, we will draw the attention to extension with other integration schemes. This is in practice tightly connected to *the constraint solver*. The integration schemes were discussed in section 3.6. Even though some flexibility is provided for definition of new integration schemes, this will in practice have some limitations due to the design of the overall simulation loop, which has been designed with the Heun integration scheme in mind. Recall that integration of a body is in general performed in two steps. In the first step, positions are updated to $\mathbf{q}^*_{n+1}$ along with an optional prediction of the velocities, $\dot{\mathbf{q}}^P_{n+1}$. In the second step, the final velocity update, $\dot{\mathbf{q}}_{n+1}$, is performed. The benefits of the explicit Euler and Heun schemes are that the forward dynamics is linear in $\mathbf{f}_{n+1}$. This also makes the inverse dynamics linear in $\mathbf{f}_{n+1}$. Our constraint solver needs a linear relationship. Higher-order integration schemes can be implemented. However, this does require that the integrator provides an approximated linearisation for use in the constraint solver. Notice that now there can be a deviation between the forward and backward dynamics. The forces found will not cause zero relative velocity in the constraints when integrated using the higher-order scheme. However, such a higher-order scheme may still give superior results, and further research should be done in this regard. For use with a higher-order scheme, a new constraint solver algorithm that performs new linearisations during the iterative search for a solution should be implemented.

6.5.4 Development of new Collision and Constraint Solvers

Implementation of a new collision solver will allow more detailed modelling of many simultaneous collisions in larger systems of connected bodies. Future research in this area is very much needed, and the simulator provides possibility for extensions that model collisions differently. However, notice that the bodies must be non-colliding after collision solving. Otherwise, the following components might cause the overall simulation to fail.

The contact and constraint solver can similarly be exchanged with a different paradigm for inverse dynamics with contacts and friction. For instance, this could be the standard LCP solver or articulated dynamics algorithms. It is even possible to implement a solver, which is a hybrid of the articulated approach, the LCP approach and our approach. Notice that exchanging high-level components of the engine might also require different definitions in the underlying components, such as friction, constraints and integration schemes. Hence these are a bit more challenging. The modules shown on the right in figure 6.2 are modules, which are simple enough for the average user to customise. The collision and constraint solvers are a lot more advanced, and custom implementation is not a trivial task.

This concludes our discussion of the ways we currently allow our engine to be extended. We find these considerations very important for a simulator, which is intended for research. In practice, the overall simulation algorithm builds on certain assumptions that make such abstractions difficult. However, we have created interfaces that allow a maximum of customisation under the constraints posed by our design choices of the overall simulation loop.

6.6 Debugging & Data Extraction

An extendable framework for hierarchical logging of engine data is used to ease debugging and to allow extracting advanced internal information from the engine for test and benchmarking purposes. The purpose of the framework is to define data structures that are serialisable and are generic, such that different engines can be modified to use the same structure in the future. Different engines will use specific internal structures, which can not be generalised, hence, the system must be extendable. Besides the core logging facility, an extendable graphical layer is provided on top. First, the core logging facility is considered.

6.6.1 The Logging Facility

Figure 6.3 gives an overview of the software structure.

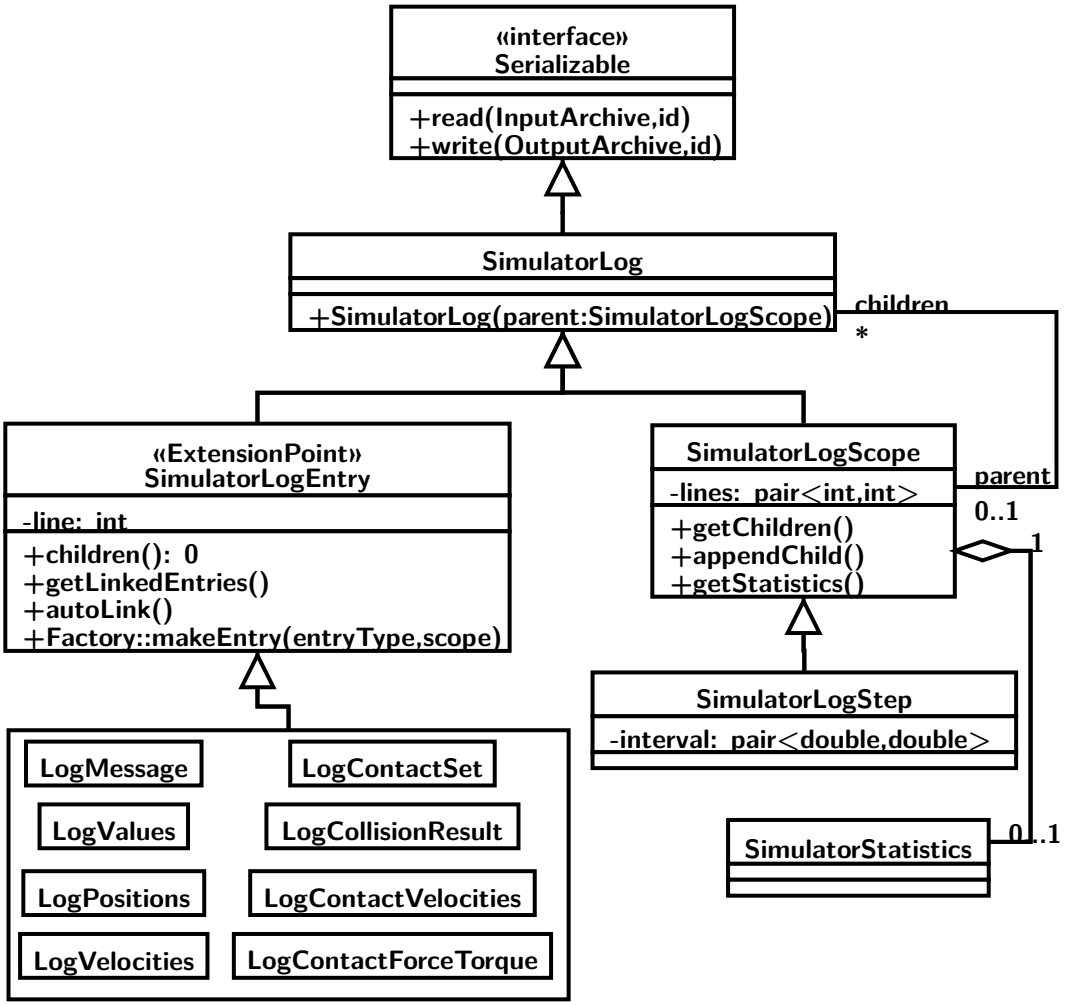


Figure 6.3: The basic logging structure.

A logging tree is implemented by using the composition pattern. A SimulatorLogScope is the root logging structure with children that are either other log scopes or log entries. A log entry is an atomic entry in the log which can have no children. A specific type of scope is a SimulatorLogStep, which is used for a simulation step and stores information about the time interval of the step. The entry type defines an ExtensionPoint as discussed in section 6.5. This makes it possible to extend the logging facility with more entry types than the default logging types. The default entry types are as follows:

Message The simplest possible log entry, which only stores a string with an arbitrary message.

Values Allows storing a list of numerical values, each with a label. This format is important in relation to the automatic statistics generation.

Positions Stores a map of a frame name to a transformation.

Velocities Stores a map of a frame name to a velocity screw of the linear and angular velocity. For visualisation purposes, this type of entry can be linked to a position entry.

Contact Set A contact set stores the frame names in contact, the contact point on both objects, the contact normal and depth.

Collision Result Stores detailed information about a collision query. This entry links to a **Positions** entry for visualisation of, for instance, triangles in collision. Basic information includes the name of the frame pair in collision and the name of the geometries in collision. However, more detailed information can also be logged, such as a list of triangle pairs in contact, and the number of bounding volume and primitive tests performed.

Contact Velocities The linear velocities of contacts. This entry must be linked to a contact set entry for visualisation purposes. The linear velocity of a contact is stored as two vectors, one for each point in the contact pair.

Contact Force & Torque Similar structure as the contact velocities entry. This entry stores the contact force and torque for each point in the contact pair. A wrench screw is stored for each point.

6.6.2 Serialisation

The RobWork serialisation framework provides a common interface for serialisation to files. Note that all log entries and scopes must implement the *Serializable* interface, which will allow logs to be saved in so-called Archives. InputArchives are used to load a serialised instance, while OutputArchives are used to save a serialised instance. Implementation of the serialisation interface allows logs to be very easily stored after a simulation, and the logs can easily be included in a bug report. Consider the example in listing 6.10. In lines 1 and 2

```
1 SimulatorLogScope::Ptr log = ownedPtr(new SimulatorLogScope());
2 engine->setSimulatorLog(log);
3
4 // Simulation...
5
6 INIArchive archive("log.ini");
7 archive.write(*log,"Simulation");
8 archive.close();
```

Listing 6.10: Running a simulation with logging.

the logging root is created and set in the engine. During simulation, the engine will add detailed debug information to the logging structure. After simulation, the log is serialised to an ini-file.

6.6.3 The Graphical Visualisation

Given the dynamic workcell and the log file, the *SimulatorLogViewer* gives a complete picture of what happened internally during simulation. In figure 6.4, an example of the log-viewer is shown. The left pane gives the hierarchical structure of the log. Notice that we are looking

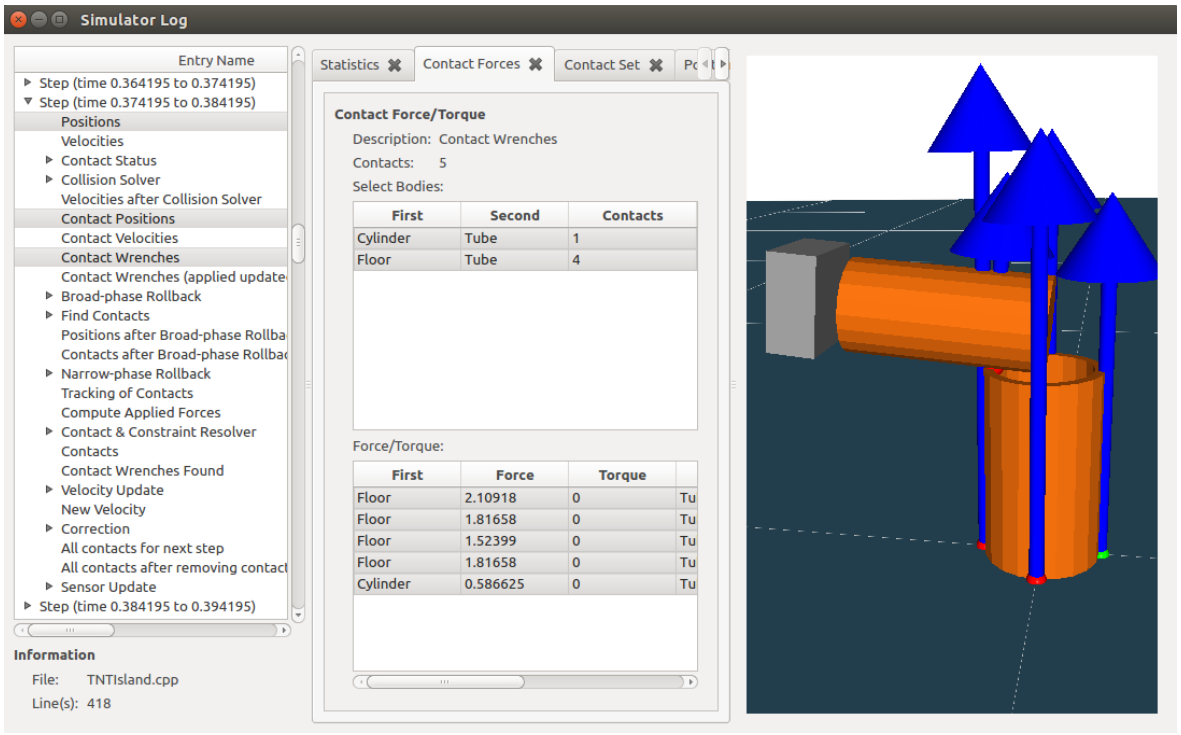


Figure 6.4: The graphical user interface for inspection of the simulator log.

at step n at $t_n = 0.374195$, which is the step after the cylinder and tube first make contact. We have chosen to show the initial body positions and persistent contacts. Furthermore, the contact wrenches are shown. In the middle pane, a tab is created for each selected log entry. In this example, the graphical interface for the force/torque entry is shown. At the top we see five contacts in total, and the body-pairs in contact are shown underneath. Individual pairs can be selected to reduce the number of contacts shown in the right pane. We see that there is one contact between the cylinder and tube, and four contacts between the floor and tube. Underneath the selection of body-pairs, all contacts are listed for the selected body-pairs. Here, the force and torque are specified. Notice that in the left bottom corner, information is provided about the source file and the line(s) where a given log entry was created. The green and red contacts shown in the visualisation indicate whether the contact is strictly penetrating or non-penetrating. In contact set view (not shown), the size of this penetration can be shown to be of the magnitude 10^{-7} m. The order of the log entries is similar to the simulation loop shown in section 3.13.

Debugging of physics engines is known to be notoriously difficult. The debugging part of the RWPE is a very important tool, which in practice makes it much easier to find the

root cause of a simulation problem. Furthermore, the dynamic workcell and log provides all information needed for a bug report. It only takes 5 lines of code to generate the information. Notice that for two seconds of simulation with a time-step $\Delta t = 0.01$, the size of the generated log file is 1.1 MB. Hence, debugging should not be enabled for large scale simulations.

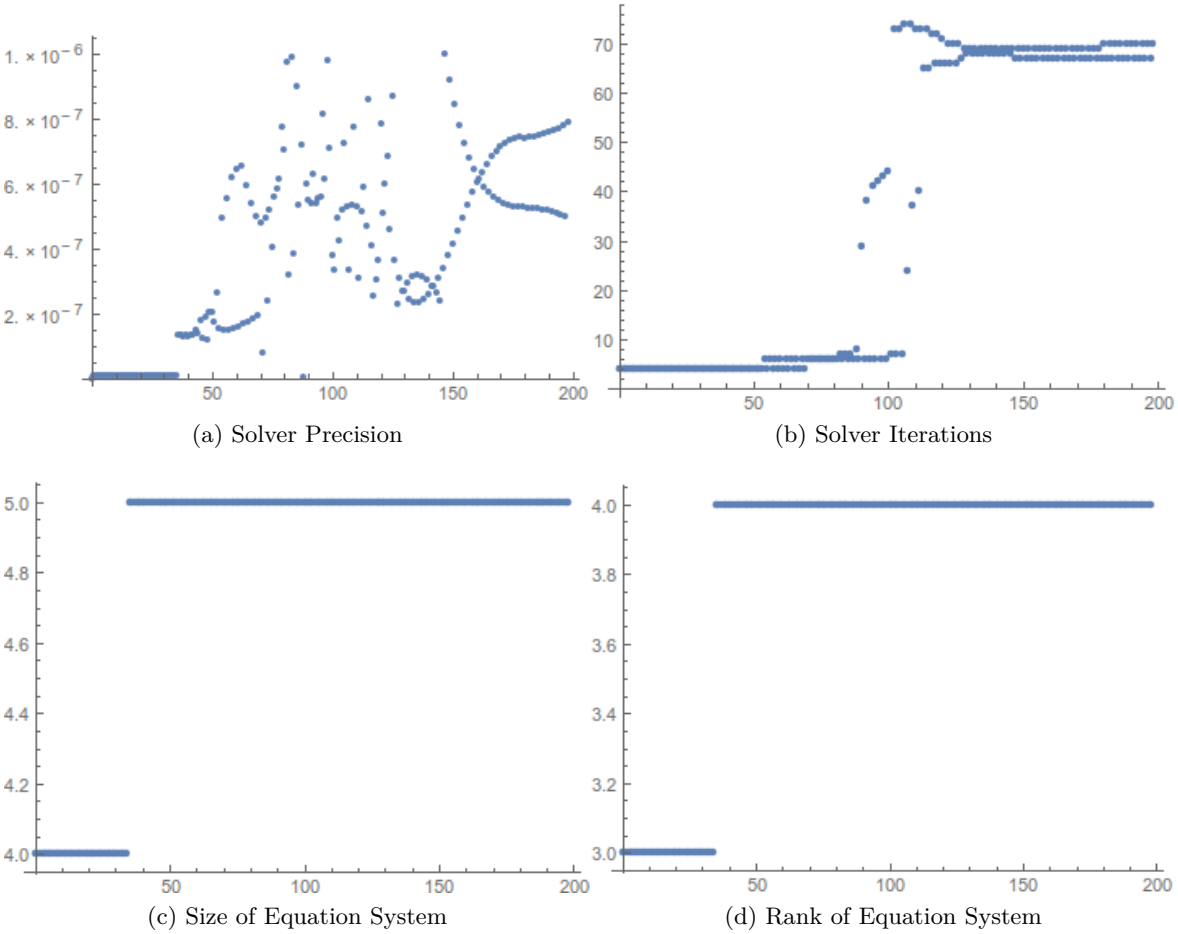
6.6.4 Extendable Logging Facility

The RWPE dynamics plugin for RobWorkSim provides an extension for the PhysicsEngine extension point, as already discussed. Besides this, it also provides an extension to the logging facility. Notice that the *SimulatorLogEntry* type defines an extension point. A unique feature for RWPE is the contact tracking method. If tracking fails, the methods can lead to problems that are difficult to debug. For this reason, the logging facility is extended with a specific RWPE tracking type. This extension allows storing information about how contacts are tracked from one point in time to another. For this to be useful, a graphical widget is also provided that extends the log-viewer. This widget is also an extension point.

6.6.5 Automatic Statistics Generation

We would like to draw attention to a useful feature in the log-viewer. It is possible to automatically generate statistics based on numerical values, which are logged by the engine through the use of the *LogValues* entry. In figure 6.3 it is shown that each log scope can have an associated statistics object. This means that the statistics are hierarchical similarly to the log tree. When numerical values are logged in the simulator, these are stored in the *LogValues* with a label that should be unique. The idea is that these values are propagated up through the tree to its parents, and that the values are handled at the appropriate level. If a log scope has log values as child entries, then each label is expected. If there are multiple log values with the same labels, the values are collected as one data series. For labels that have no matching values, the values are propagated to the parent scope to be handled at a slightly higher level. In the log viewer, a widget is provided for visualisation of automatically generated statistics.

We can consider a few of the statistics collected at the highest level in figure 6.5. In



(a) Solver Precision

(b) Solver Iterations

(c) Size of Equation System

(d) Rank of Equation System

Figure 6.5: Automatically generated statistics from the simulator log facility.

figure 6.5a, the obtained error criteria for the linear solver is shown. This is the size of $\|\Delta \mathbf{s}_X\| + \|\Delta \mathbf{s}_B\|$ from the gradient of the objective function in equation 3.95. Notice, however, that in this case the value has been normalised against the minimum constraint force in the system to make the measure dimensionless. Figure 6.5b shows the corresponding number of iterations used to obtain the solution with desired accuracy. When the cylinder and tube make contact, the number of iterations go up. The size of the equation system, 3.88, increases from 4 to 5 as shown in figure 6.5c. Notice, however, that the rank, K, is always one less than the system size as shown in 6.5d. This is expected as the tube has four contacts with the floor. Hence, there is redundancy in the system.

6.6.6 Using Mathematica Integration for Visualisation

For visualisation of graphs, the Wolfram Symbolic Transfer Protocol (WSTP) [94] has been used through C/Link. The generation of graphs is too complex to describe here. Instead, a small example will be given of the use of the RobWork WSTP layer in general. Consider the solution of the following problem using a Wolfram Kernel:

$$x^2 + 2y^2 = 3681, \qquad x > 0, \quad y > 0 \tag{6.1}$$

In listing 6.11, an example is shown that will solve this problem. Lines 1 and 2 initialises

```
1  Mathematica m;
2  m.initialize();
3  const Mathematica::Link::Ptr l = m.launchKernel();
4  Mathematica::Packet::Ptr result;
5  *l >> result; // read the first In[1]:= prompt from kernel
6
7  const char* cmd = "Solve[x^2 + 2 y^3 == 3681 && x > 0 && y > 0, {x, y}, Integers]";
8  *l << cmd;
9  *l >> result;
10 std::cout << "Result: " << *result << std::endl;
11 if (const ReturnPacket::Ptr packet = result.cast<ReturnPacket>()) {
12    const List::Ptr list = List::fromExpression(*packet->expression());
13    const std::list<rw::common::Ptr<const Mathematica::Expression> > sols =
          list->getArguments();
14    BOOST_FOREACH(const rw::common::Ptr<const Mathematica::Expression> sol, sols) {
15       const rw::common::Ptr<const Mathematica::FunctionBase> fct = sol.cast<const
             Mathematica::FunctionBase>();
16       const PropertyMap::Ptr map = Rule::toPropertyMap(fct->getArguments());
17       const int x = map->get<int>("x");
18       const int y = map->get<int>("y");
19       if (x*x+2*y*y*y == 3681 && x > 0 && y > 0) {
20          std::cout << "Found solution: x=" << x << " and y=" << y << std::endl;
21       }
22    }
23 }
```

Listing 6.11: Using the RobWork WSTP interface.

the underlying WSTP C-framework. In line 3, a new kernel is launched and a handle is returned for the WSTP link to the kernel. The first packet on the link is retrieved in lines 4 and 5. This is basically a query from the kernel for the first expression. In lines 7 and 8 the command is send. This is just specified as a string similar to the command one would use directly in Mathematica. The command is sent in a packet in line 8, and in line 9 the resulting packet from the kernel is retrieved. Line 10 will in this case result in the output shown in table 6.1. The returned packet is of type ReturnPacket, so in line 11 the returned Packet is cast. In line 12, a RobWork equivalent of a List expression is constructed directly from the packet expression. The arguments of the list correspond to the three inner lists all of which are a solution to our problem. In lines 13 to 14 we iterate over the list, and in line

```
Result: ReturnPacket[
   List[
      List[
         Rule[x, 15],
         Rule[y, 12]],                    Found solution: x=15 and y=12
      List[                               Found solution: x=41 and y=10
         Rule[x, 41],                     Found solution: x=57 and y=6
         Rule[y, 10]],
      List[
         Rule[x, 57],
         Rule[y, 6]]]]
```

Table 6.1: Example of the result of a Mathematica operation solved using the RobWork WSTP system.

15 the inner list is cast as a FunctionBase, which is a generic RobWork type that can refer to any Mathematica function. The arguments are the two Rule elements that are converted to a RobWork PropertyMap in line 16. The x and y values can then be extracted from the PropertyMap in lines 17 and 18. Lines 19 to 21, result in the output shown on the in table 6.1.

This concludes the discussion of the simulator logging facility. The facility provides a nice environment for debugging of the engine that is easy to use for debugging of the engine. It also allows easy extraction of internal parameters, which will make parameter tuning much more easy. In the future, this system can be used for developing applications that will allow very user-friendly guided tuning of parameters.

6.7 Multi-Threading in Dynamic Simulation

Using dynamic simulation in robotics has different objectives. Typically, it is used for offline learning of optimum strategies, where large numbers of experiments are performed in batch-mode. In this case, there are no real-time requirements. In other cases, dynamic simulation can be a tool to learn parameters in an online system, and in this case the simulation should run fast and efficient. Previously work has been done on parallelisation of ODE [95, 96, 97]. This work focuses on breaking down the dynamics into smaller sub-systems. In our case, an iterative method is used, and we believe that trade-offs between speed and accuracy should be based on tuning of the objective functions. However, we do find it important to do work in parallel if it can be done in parallel. The engine should be able to utilise the resources it is given, but it should also respect limits in resources, such that it does not clog up a critical system. The objective is to make it easy for the user to choose any number of threads that should be used for simulations. It should allow reusing the same code, with a minimum of extra implementation making it parallelisable. We will use the thread pool pattern to achieve this goal. We will present the pattern before considering a hierarchical task format, which is the main contribution to the framework.

6.7.1 The Thread Pool Pattern

The Boost Asio allows cross-platform, asynchronous programming. This is used as a basis for implementing the thread pool pattern and allows the user to specify the number of threads to use, n, which will be the size of the pool. Work can then be assigned to the pool where it will be queued and processed by the first available thread. To make an unified programming interface, the pool has been designed to allow specifying $n = 0$, which will simply execute the work in the invoking thread by making the AddWork function synchronous. For $n > 0$, the AddWork function will add the work to the queue and return asynchronously.

The main benefit of the thread pool pattern is that it allows the user to control the number of threads to allocate for running one or more simulations. If a large number of simulations need to be run for offline learning, the maxmimum number of cores in the system can be utilised. If, on the other hand, the simulation must run on a real platform or with real-time user interaction, the amount of resources used by simulation can be limited to allow other processes to perform critical work.

6.7.2 A Hierarchical Task Format

Using the hierarchical task format allows each computation task to divide itself into multiple subtasks, which can again be executed in parallel. The subtasks are executed in parallel if enough threads are available in the pool. If there are not enough threads, the task will have to wait until a thread becomes available. By using the hierarchical task format, the allocated resources will be exploited as much as possible.

The definition of computation task now involves implementation of the following methods:

Run – the main work unit of the task. In this function, new subtasks can be added. This will allow the task to branch out in multiple parallel computations.

SubTaskDone – invoked when a subtask has ended. It allows using the result of a subtask to either launch more subtasks or create a combined result of subtasks performed so far.

Idle – invoked if Run and all subtasks have ended. This is the last chance to do more work by adding new subtasks. The results of all subtasks can be combined at this step.

Done – invoked when all work has been done. It is no longer possible to add more work, and the task will finish immediately after this invocation. This might in turn cause a invocation of the Idle function in the parent task.

Each task can be in a number of different states as shown in the state diagram in figure 6.6. The task is initially in the *Initialisation* state. In this state, the task has not yet added its

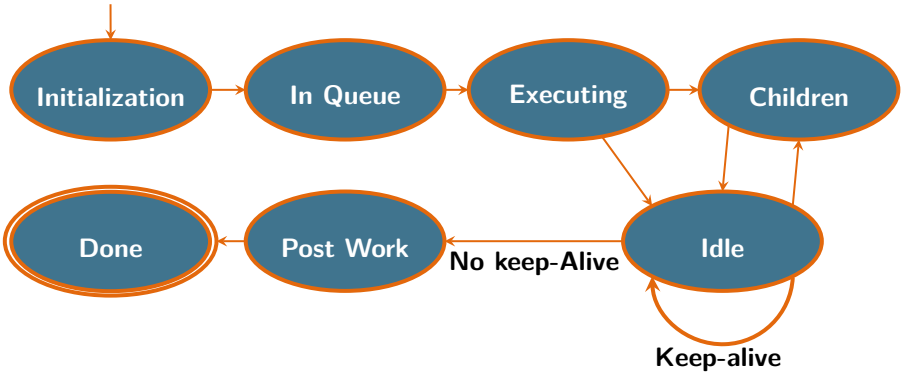


Figure 6.6: The finite state machine for a computation task.

work to the thread pool work queue. If the task is added as a subtask to an already running task, the task will transition into the *In Queue* state. This means that the task has put itself into the worker queue and is now waiting for a thread to become available. For a root-level task, the task must initially be launched by calling the EXECUTE function, which will also launch all subtasks in the hierarchy. When a thread becomes available, the task transitions into the *Executing* state, and the code in the RUN function is now running. If this task is a leaf task and has not launched any subtasks, the state will transition to *Idle* when the RUN function exits. If subtasks have been launched, the task transitions into the *Children* state which indicates that the task is waiting for subtasks to finish before proceeding. When each subtask gets to their *Post Work* state, the SUBTASKDONE method is invoked. This allows launching even more subtasks if required. At the time where the last subtask has finished, and the SUBTASKDONE function has not added any more subtasks to be executed, the task will transition into the *Idle* state. Now, the IDLE function will be invoked allowing one last change to perform more work by adding subtasks. If a keep-alive flag is set, the task will not finish itself before the flag is cleared. Otherwise, the task will proceed to the *Post Work* state, which will invoke the DONE function and then the SUBTASKDONE function in the parent task to notify the parent that this task is now finishing. When this has been done, the task transitions to the *Done* phase to indicate that all work has been done.

Note that this hierarchical task format allows refining tasks on many levels while providing a interface, which makes it easy to define *branch and combine* types of tasks. An example of a task hierarchy is shown in figure 6.7. In this example, a set of assembly simulations should be run which can of course be done in parallel as shown. In a batch assembly simulation, it is unlikely that there is more than one simulation world. If, on the other hand, the user was running another type of simulation, it might be possible to simulate multiple worlds independently. For each world, there can be a dynamic division in islands where the dynamics can be computed independently. The example shows the use of parallelisation for the Hybrid type of collision solver described in section 3.5. Here, it might be possible to construct

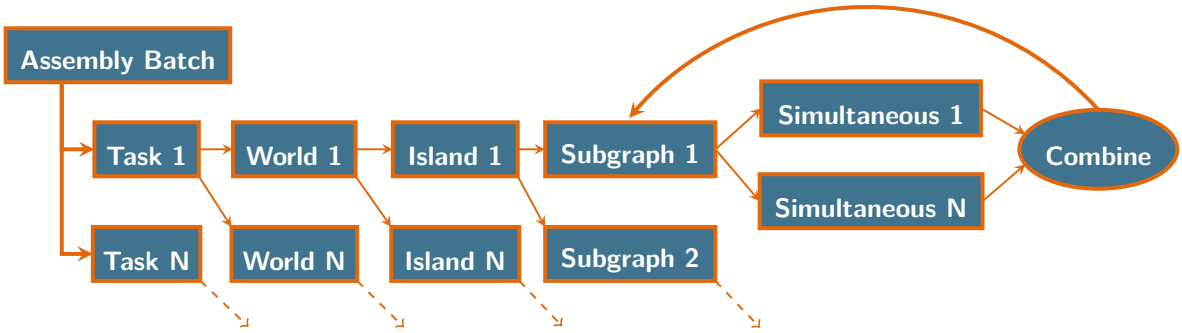


Figure 6.7: Example of a hierarchic task decomposition for maximum parallelisation.

sub-graphs, which are independent for collisions, allowing parallelisation. For each of these sub-graphs, an iterative algorithm is used to handle small sections of simultaneous impulses and combines the result. Based on the result, new collisions might occur, and a new round is done with a new set of parallel simultaneous solvers. This branch and combine behaviour is facilitated by the designed task format.

Note again, that different use cases require parallelisation at different levels. If doing batch assembly simulation, parallelisation at the lowest level of impulse handling will probably not cause a large gain in performance, unless many cores are available. If, on the other hand, a single island simulation is performed for real-time purposes, the decomposition in parallel tasks might be able to speed up the simulation in general.

6.7.3 Exception Handling

Exceptions are somewhat difficult to handle in this hierarchical approach. The exceptions can not be propagated as they normally would due to threading. Instead, all task implementations should be careful not to throw any exceptions. These must be catched and registered in the task. The SubTaskDone function should then take explicitly care that the exceptions are handled or propagated properly. Due to this, the default implementation of SubTaskDone will register all exceptions in subtasks in the parent task. If this function is overridden by the user, the user should implement a similar functionality.

This concludes the discussion on parallelisation in robotics simulation with focus on flexible resource allocation. This is achieved through an implementation that can utilise as many cores as possible within the limits given by the user.

6.8 Developing Control Strategies in MatLab

An interface has been developed for use of RobWork from MatLab. SWIG [92] is used for automatic generation of a Java interface, through which MatLab can interact with RobWork. A Java library is generated for RobWork, RobWorkSim and RobWorkStudio respectively. Each of these use the Java Native Interface (JNI) to call the native C++ functions in RobWork. Notice that when launching MatLab, the Java Classpath must include these three jar files. In listing 6.12, an example is shown of a piece of MatLab code that launches RobWorkStudio, loads a Dynamic WorkCell, and exits again. In lines 1 to 4, the native libraries are loaded by

```
1  import dk.robwork.*;
2  LoaderRW.load('RobWork/libs')
3  LoaderRWSim.load('RobWorkSim/libs')
4  LoaderRWS.load('RobWorkStudio/libs')
5
6  rwstudio = rws.getRobWorkStudioInstance();
7  dwc = DynamicWorkCellLoader.load('Scene.dwc.xml');
8  wc = dwc.getWorkcell();
9  state = wc.getDefaultState();
10 rwstudio.postWorkCell(wc);
11
12 % Main Content
13
14 rwstudio.postExit();
```

Listing 6.12: Launching RobWorkStudio and loading a dynamic workcell through the MatLab interface.

some convenience functions. RobWorkStudio is launched in line 6, and a dynamic workcell is loaded in line 7. The workcell and state can then be retrieved from the dynamic workcell in lines 8 and 9. In line 10, the workcell is loaded into RobWorkStudio, and finally in the last line RobWorkStudio is closed again.

Now, the interesting part of the interface, is using it for running a simulation. In listing 6.13 an example is shown of how a simulation can be initiated from MatLab. The procedure is somewhat similar to the simulation example in section 6.3. In lines 1 to 3, the engine and dynamic simulator are created. In this example, we choose to use the *ThreadSimulator*, which will run the engine in a separate thread. This type of simulator makes asynchronous callbacks at each simulation step. In lines 8 to 17, the callback functionality is implemented, which is somewhat complex. Notice that in line 11 the MatLab callback, is listed in 6.14, is created. In line 18, the simulator is started and in line 22, the simulation can be aborted.

The MatLab callback function is shown in listing 6.14. These lines of code will set update the RobWorkStudio visualisation every 20 times the StepCallBack function is called from RobWork. Also, the current simulation time will be printed in MatLab. This is implemented in lines 13 to 17.

```
simulator = PhysicsEnginePtr(RobWorkPhysicsEngine(dwc));
simulator.initPhysics(state);
dsim = DynamicSimulatorPtr(DynamicSimulator(dwc, simulator));
tsim = ThreadSimulatorPtr(ThreadSimulator(dsim, state));
tsim.setTimeStep(0.001);

%% Create callback and run Simulation
dispatcher = ThreadSimulatorStepEventDispatcher();
setappdata(dispatcher,'UserData',[0]); % set step counter
% Set the callback (rwstudio pointer is passed as additional argument)
set(dispatcher,'StepEventCallback',{@StepCallBack,rwstudio});
% Manually invoke callback (just a test)
dispatcher.callback(tsim.dereference(),state);

% Add to simulator and start
fct = ThreadSimulatorStepCallbackEnv(dispatcher);
tsim.setStepCallBack(fct);
tsim.start();

%% Exit RobWorkStudio
if (tsim.isRunning())
    tsim.stop();
end
```

Listing 6.13: Launching dynamic simulation from MatLab.

```
function StepCallBack( dispatcherObject, event, rwstudio )
    import dk.robwork.rw;

    % Extract the event data
    tsim=event.getThreadSimulator();
    state=event.getState();

    % Extract user data stored on dispatcher object
    userdata=getappdata(dispatcherObject,'UserData');
    counter = userdata(1);

    % Set state in RobWorkStudio and print time for each 20 steps
    if mod(counter,20) == 0
        rwstudio.setState(state);
        display(num2str(tsim.getTime()));
        rw.writelog(strcat(num2str(tsim.getTime()),'\\r\\n'));
    end

    % Update step counter
    setappdata(dispatcherObject,'UserData',[counter+1]);
end
```

Listing 6.14: Asynchronous MatLab callback for each step of the simulator.

Our motivation for doing an integration with MatLab, is similar to the motivation behind implementation of a physics abstraction layer. It will make it easier for users, which are not familiar with RobWork, to use the functionality provided by RobWork. For development of controllers, MatLab is one of the standard tools. Hence, for development of controllers for assembly, we find it important to have this functionality.

6.9 Test Framework

A unified system for defining standard tests for engines is implemented. Once again an extension point is provided called *EngineTest*. Tests are supposed to be generic with respect to the chosen engine. However, in some cases, there can be reasons that a specific test only makes sense to run specific engines. The test will return with a set of results. Each of the results are a dataset of importance to the test. Furthermore, the test allows specification of a parameter map as input to adjust the behaviour. In figure 6.8, the RobWorkStudio plugin for engine testing is shown. In 6.8a, the engine and test are chosen from the extension library. Notice that the list in both the left and right pane can be extended through plugins. In 6.8b, different parameters can be set to tune the test. This set of parameters is supposed to be as small as possible to capture only significant parameters, which are important for the specific test. In figure 6.8c, the test is executed, and a list of different results is shown. In this case, it is possible to plot forces, timing and distances. An example of the latter is shown in figure 6.8d.

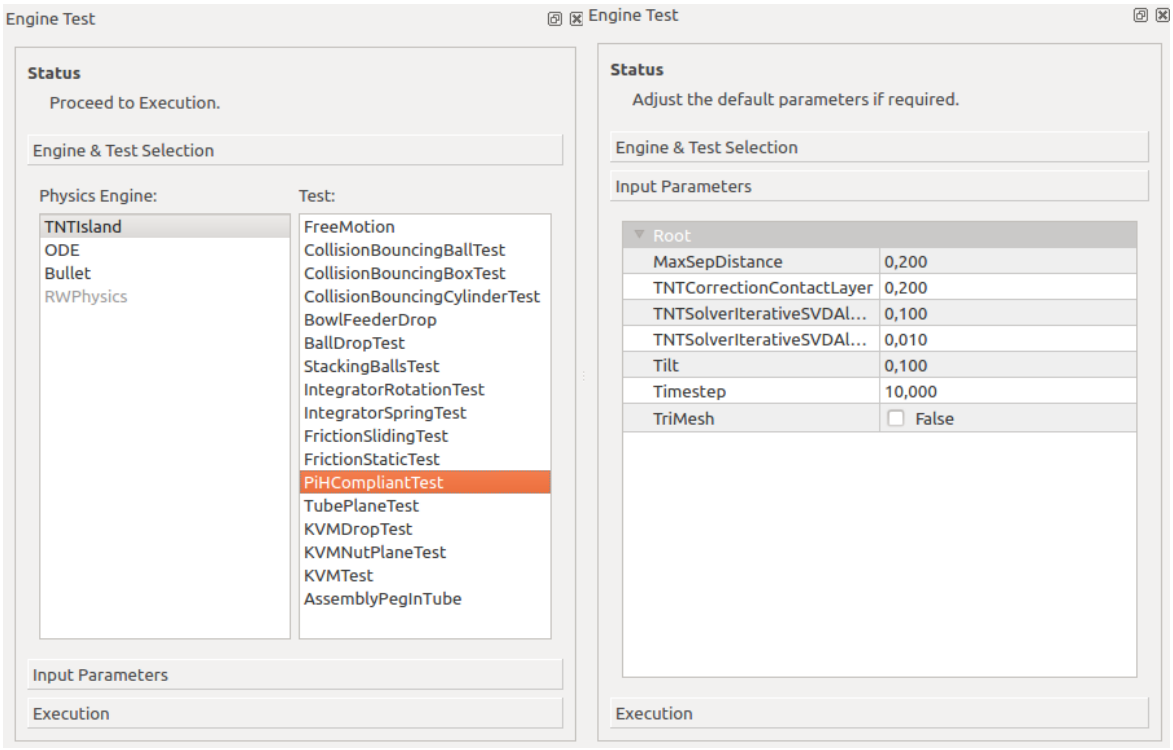
The main purpose of this part of the framework is that the exact same tests can be reused for different purposes:

Unit testing Continuous automated testing of the code-base is important. Especially for a complex piece of software as the RWPE. By running tests that focus isolated on individual parts of the engine, it will be easier to sport errors that occur during daily development.

Interactive testing in RobWorkStudio The plugin provides a simple way to quickly run small standardised tests. Through the debugging facility it is possible to get detailed information on a potential problem.

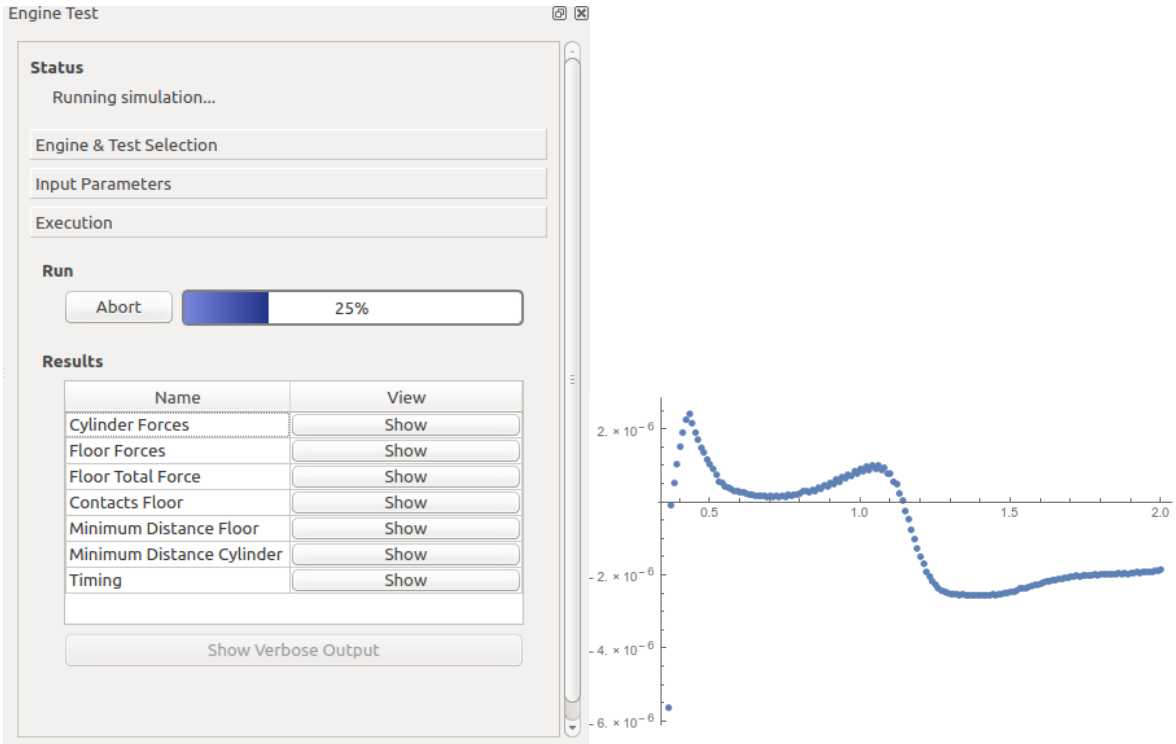
Benchmarking We see benchmarking as a different form of test than the above two cases. The purpose of a benchmark is to stress-test the engine and gather information about performance. All unnecessary debugging output should be disabled, and the engine should be compiled in a special optimised build. In qualitative testing, simpler tests are performed and we need as much logging as possible to understand in detail what is going on.

Building the test framework is ongoing work. The main purpose is to create one unified way of specifying tests for the above three cases.



(a) Engine and test selection.

(b) Parameter choice.



(c) Simulation and results.

(d) Minimum distance of cylinder.

Figure 6.8: An example of running a test using the *EngineTest* plugin for RobWorkStudio.

CHAPTER 7

Using the Engine for Assembly

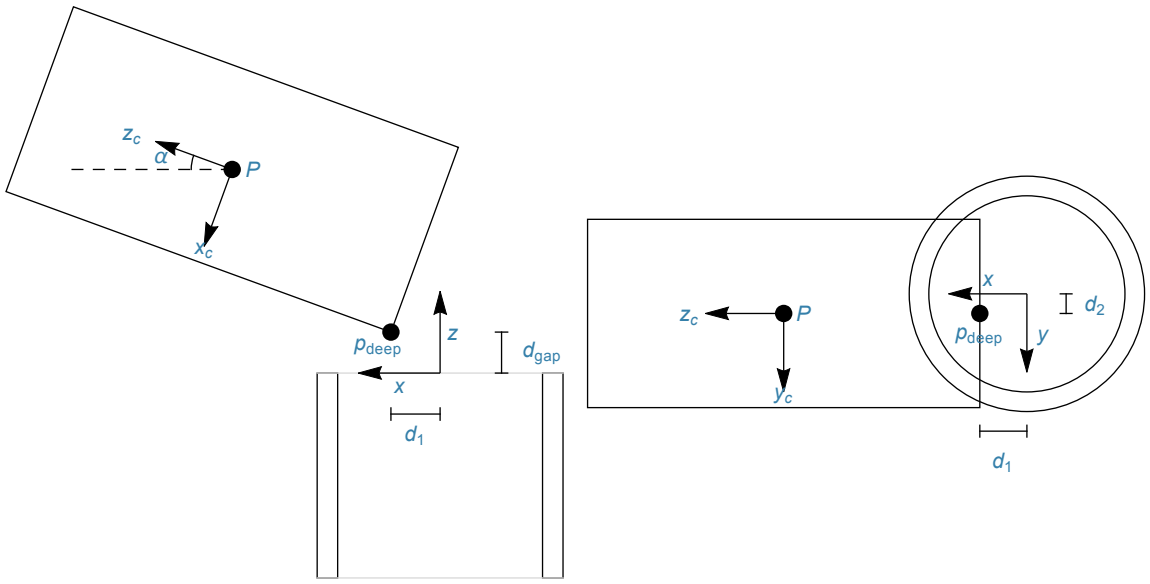
The main purpose of the developed engine is simulation of assembly actions. The framework for assembly simulation is now presented as a layer on top of the engine. The idea of this approach is to allow the use of different engines. First of all, the user interface for the assembly simulator is presented, and subsequently the assembly simulator itself is presented. The main purposes of the assembly framework are:

- Development of strategies that are completely unaware if they are being used in a simulated or real environment. This facilitates code reusability.

- Preventing the user from using information that will not be known in a real setup. It is tempting to use information from simulation that would not be available under normal circumstances, as the simulator provides complete knowledge of the state of the entire system. The control strategy must use information from sensors only. These sensors are defined with the same interface for real and simulated sensors.

- The core task of an assembly operation is to place two objects relative to each other. The strategy should work on relative positions and should not know about a global reference frame. If for instance both bodies can be moved, the assembly strategy should not be responsible of placing the two bodies in absolute coordinates. Instead the strategy should provide the desired relative placement of the bodies. Absolute placement is then a task for a lower level controller. This will make the same optimised assembly strategies available in different setups. If for instance one hand is suddenly fixed instead of actuated, this should not influence the assembly strategy itself.

In the following the different concepts in the assembly simulator will be introduced. An example of a simple peg-in-hole strategy will be given to illustrate these concepts. The assembly action is seen as an atomic operation, and the purpose of the assembly task format is to create a somewhat standardised way of specifying, serializing and visualizing an assembly action. In figure 1.1 and 6.1 peg-in-tube examples are shown with two bodies that should be assembled by an ordinary insertion approach. Throughout the chapter this peg-in-hole task will be used as an example.



(a) Side view (b) Top view

Figure 7.1: Example of parameterisation of a circular peg-in-hole task.

7.1 Example of an Assembly Strategy

For a peg-in-tube action we propose a simple strategy. For reference, a different strategy was proposed and optimised in [98] for the same task, using the same assembly simulation framework that we present here. First step of the definition of a strategy is to do a parameterisation of the problem. The approach pose $\mathbf{T}_{male}^{female}$ should be given directly from the parameterisation. In figure 7.1 the parameterisation is illustrated for a hole with radius R, and a cylinder of length l and radius r. Note that it is required that $r < R$. The reference frame is the $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ coordinate system defined at the top of the hole where the peg is to be inserted. The parameters used to parameterise the action, are d_1, d_2 and α. The distances d_1 and d_2 is illustrated in figure 7.1b, and defines the location of the deepest point, $\mathbf{p}_{deep}$, relative to the center of the hole. The distance d_1 is the distance along the direction z_c of the cylinder projected onto the hole plane, and d_2 is the distance parallel to d_1. The angle α defines the initial tilt of the peg relative to the hole. Finally the gap size, d_{gap}, is a fixed value.

The three parameters will be sufficient to construct a unique approach pose, $\mathbf{T}_{female}^{male}$. The procedure is as follows:

1. Determine the desired $\mathbf{p}_{deep} = d_1\mathbf{x} + d_2\mathbf{y} + d_{gap}\mathbf{z}$, where $\mathbf{x}$ and $\mathbf{y}$ are arbitrary orthonormal vectors lying in the hole plane. The direction of the cylinder, $\mathbf{z}_c$, will be based on the direction $\mathbf{x}$.

2. Determine the cylinder direction $\mathbf{z}_c = \mathbf{R}_{EAA}(-\alpha\mathbf{y})\mathbf{x}$.

3. Determine $\mathbf{x}_c = \frac{\mathbf{y}\times\mathbf{z}_c}{\|\mathbf{y}\times\mathbf{z}_c\|}$ and $\mathbf{y}_c = \frac{\mathbf{z}_c\times\mathbf{x}_c}{\|\mathbf{z}_c\times\mathbf{x}_c\|}$.

4. Find the position of cylinder, $\mathbf{p} = \mathbf{p}_{deep} + \frac{l}{2}\mathbf{z}_c - r\mathbf{x}_c$.

5. The approach pose $\mathbf{T}^{male}_{female}$ is now given directly from $\mathbf{p}$, $\mathbf{x}_c$, $\mathbf{y}_c$ and $\mathbf{z}_c$.

The parameters to be optimised will be d_1, d_2 and α. The parameter space of interest is $d_1^2 + d_2^2 \leq R^2, d_2 > 0$ and $\frac{\pi}{2} \geq \alpha \geq 0$. Notice that even though the strategy suggests this approach pose, it is completely unknown whether or not this approach pose is actually achieved.

The main part of an assembly strategy is the control loop for the insertion phase. In this strategy, the insertion phase will be divided into five states which is executed in sequential order. Notice at this point that no assumption is made on how the peg and tube are controlled. They could be attached to a kinematic body through a compliant connection or could be grasped by a manipulator.

Approach It is assumed that the system is initially at rest and that the relative pose is somewhere close to $\mathbf{T}^{male}_{female}$. At this point a reference force/torque, $\mathbf{f}_{ref}$ and $\mathbf{t}_{ref}$, is stored for later. The peg is then moved with a constant linear velocity $-v^{male}_{female}\mathbf{z}_a$ in the opposite direction of the assumed hole direction z. It is assumed that the magnitude of the velocity is small enough that the force and torque measurements will settle close to $\mathbf{f}_{ref}$ and $\mathbf{t}_{ref}$ before the bodies get into contact. The measured force is monitored during approach and when the change in force exceeds a threshold $\|\mathbf{f}_m - \mathbf{f}_{rel}\| > \Delta f_{limit}$ the strategy proceeds to the rotation phase. It is then assumed that the two bodies have come into contact.

Rotation In the rotation phase the bodies are rotated relative to each other in order to align $[\mathbf{z}_c]_a$ and $\mathbf{z}_a$. It is an objective to have as little interaction force between the peg an tube as possible. At the same time we want to move the bodies relative to each other such that they maintain contact. Knowing if bodies remain in contact is however a difficult problem. The only physical sensor that will give useful information is the force/torque sensor. Our strategy should rely on force/torque information only, to guide the peg into the hole. Notice that it is typically mounted at the end of an robot arm in-between the arm and the manipulator. It is uncertain how the peg is grasped by the manipulator and the grasp is compliant. Determining the contact force on the cylinder based on the force/torque reading is now very difficult. The mass distribution might change during insertion if for instance the peg slides in the hand. A method is needed that can infer the position of the peg in the hand as well as the contact force. We do not address this problem in our small example, but it is an interesting problem that our framework makes it possible to investigate further.

Finalise When the peg has been rotated, the peg is simply moved linearly into the hole. The action is finished, and there are no more work to be performed by the strategy.

Notice that this strategy is mostly provided as a simple example of a assembly strategy. In reality it will not be expected to be very reliable.

7.2 Definition of an Assembly Task

An example of the definition of an *AssemblyTask* is given in table 7.1. The task is composed of four main sections. In the required section the absolute minimum specification of an assembly task is given. First of all the name of the two bodies to assemble must be specified. We refer to one of the bodies as the male, and the other as the female. In the case of the peg-in-tube example, the cylinder is the male body and the tube is the female body. The goal of the assembly task is to bring the two bodies into a certain relative position. In this case we want the body coordinate systems to be aligned, and the mass centre $\mathbf{p}$ of the cylinder to be at depth $\frac{l}{2}$. The strategy is chosen as the peg-in-tube strategy, and parameters are given for insertion. An extendable assembly registry is look up the given strategy identifier. In the optional section different reference frames can be given for the two bodies. A reference frame must be specified if the strategy works in a different frame than the body frame. In this case the cylinder body frame is different than the frame defined for the hole. Finally some arbitrary meta-data can be associated to the task. In the controllers and sensors section,

Required	
MaleID	Cylinder
FemaleID	Tube
Target $\mathbf{T}^{maleTCP}_{femaleTCP}$	$-\frac{l}{2}\mathbf{z}$
Strategy	Peg-in-tube (section 7.1)
Parameterization	d_1,d_2 and α (strategy specific)
Optional	
Male TCP	
Female TCP	Hole
Meta-data	Task identifier, name of workcell, author and date.
Controllers & Sensors	
Male Pose Controller	Box
Female Pose Controller	
Male F/T Sensor	Between cylinder and peg.
Female F/T Sensor	
Simulation Specific (for extended visualisation)	
Male Flexible Frames	Cylinder, Box
Female Flexible Frames	
Body Contact Sensors	Sensor for contacts on tube.

Table 7.1: Definition of an AssemblyTask with some example values.

a controller is specified for the male body. Notice that a controller must be specified for either the male or female body (or both). This controller is any controller that is able to manipulate the object. It could be a controller for a robot arm, a manipulator or as here a kinematic body. When specifying a kinematic body the controller will automatically use the

BodyController which is described in section 3.3. A force/torque sensor can be specified per body. Specifying such a sensor will feed force/torque information to the assembly strategy. Finally some simulation specific parameters can be set. These are intended for visualisation purposes. A serial chain of frames can be specified for each body, in which case the transform of all bodies will be stored in the output after simulation. Specifying the box frame here, will make the simulator store information about the transform between the cylinder and box. Notice that we are cheating as this would rarely be known in reality. For visualisation purposes it is however convenient. The task definition has deliberately been defined at a high level of abstraction. The purpose is to make the definition of an assembly task as user friendly as possible.

7.3 An Assembly Result

After considering the assembly task in section 7.2, it is natural to continue with the corresponding result format. In table 7.2 the format is shown. Notice that the required section contains only one binary field. Either the assembly was successful or not. The final relative

Required	
Success	True or false.
Optional	
$\mathbf{T}_{femaleTCP}^{maleTCP}$	The achieved relative transform from the female frame to the male frame.
Task ID	The name of the task specification that was executed.
Result ID	An optional identifier to distinguish the result.
Real State	The real state (typically only known for simulation).
Assumed State	The state as seen by the strategy.
Approach	The used approach.
Error	Failure codes. Can for instance indicate a simulation error.
Error message	Additional error information can be stored in the result.

Table 7.2: Definition of an AssemblyResult.

pose should also be specified, if possible. If an error was encountered a failure code can be set or an arbitrary error message can be stored. Metadata can be set such that the result can be identified and related to a specific task. The approach field stores the approach that was used by the strategy. Notice that in the result a distinction between real and assumed state is used. Real state is in general unknown. Normally only simulators will store this information for visualisation purposes. The assumed state provide information about what knowledge the assembly strategy had during execution. Storing information about the assumed state will make visualisation of the controller state possible.

7.3.1 The Assembly State

The assembly state structure allows storing values of the state of the system. Assumed state will in general be required in both simulation and in reality for the strategy to work. Notice that it is entirely up to the strategy to update the assumed state according to its internal knowledge of the system. This knowledge will rely on either simulated or real sensors. The strategy should never know of the real state of the system, as this is only experienced through sensors that might be have uncertainties. In practice the interface for the strategy allows to use the real state in a simulated system. This is however considered "cheating". Values stored in the state includes the current phase of the strategy, the relative transform, measured force and torque, contact information and additional optional transforms specified in the task format.

7.4 The Control Strategy Interface

Implementation of a control strategy requires definition of some functions. First of all the two meta-data functions GetID and GetDescription must be provided to provide a useful description of the strategy. In order for the task format to be serialisable, the control strategy is responsible of defining a parameterisation. The method CreateParameterization(γ) must be implemented to construct a parameterisation based on a generic list of parameters. The GetApproach functions gives the transform $\mathbf{T}^{maleTCP}_{femaleTCP}$ based on the parameterisation chosen. Finally the core assembly control loop is implemented in the Update function:

$$\text{Update}(\gamma, \text{Real State, Assumed State, Internal State,}\mathbf{f}_{male},\mathbf{t}_{male},\mathbf{f}_{female},\mathbf{t}_{female},t_n)$$

Here the real state is given as input to the strategy. In normal circumstances the strategy should however not rely on this information. The assumed state is an input that the strategy should update to reflect its internal belief of the system configuration. The internal state allows the strategy to store internal information between calls. Finally force/torque inputs for the male and female sensors are given if specified in the task format, along with the current time t_n. The Update returns an *Assembly Control Response* which is used to request different types of control and set targets. Control modes include position, trajectory, velocity and hybrid force/torque control. Notice that the body controller chosen in the task format must implement the relevant types of control.

7.5 Assembly Simulation

In our considerations so far we have avoided making any assumptions on assembly in real or simulated contexts. The real and assumed states are important distinctions used in the assembly simulator. The assumed state is a partial and uncertain view of the real state of the

system. In assembly simulation we equate reality with the dynamic simulation, and work on deliberately reduced information. The assembly simulator works in four sequential phases:

$$\text{Initialisation} \rightarrow \text{Approach} \rightarrow \text{Insertion strategy} \rightarrow \text{Done}$$

In the initialisation phase the dynamic workcell is loaded, the assembly task specification is read and dynamic simulation is initiated. In the approach phase the simulator queries the strategy with GetApproach for an initial relative pose. The simulator moves the objects to the relative position requested, and proceeds to the insertion phase. In this phase the Update function of the control strategy is called in each iteration of the simulation. When the strategy indicates that it is done, or returns with a failure, the assembly simulation is done. The assembly result can then be retrieved from the simulator yielding detailed information about the simulation performed.

7.6 The Assembly Visualisation Plugin

In figure 7.2 the RobWorkStudio plugin for visualisation of assembly is shown. The plugin makes it easy to shift between visualisation of the real and assumed states to compare reality with the controllers view on reality (the assumed state). We believe that the developed frame-

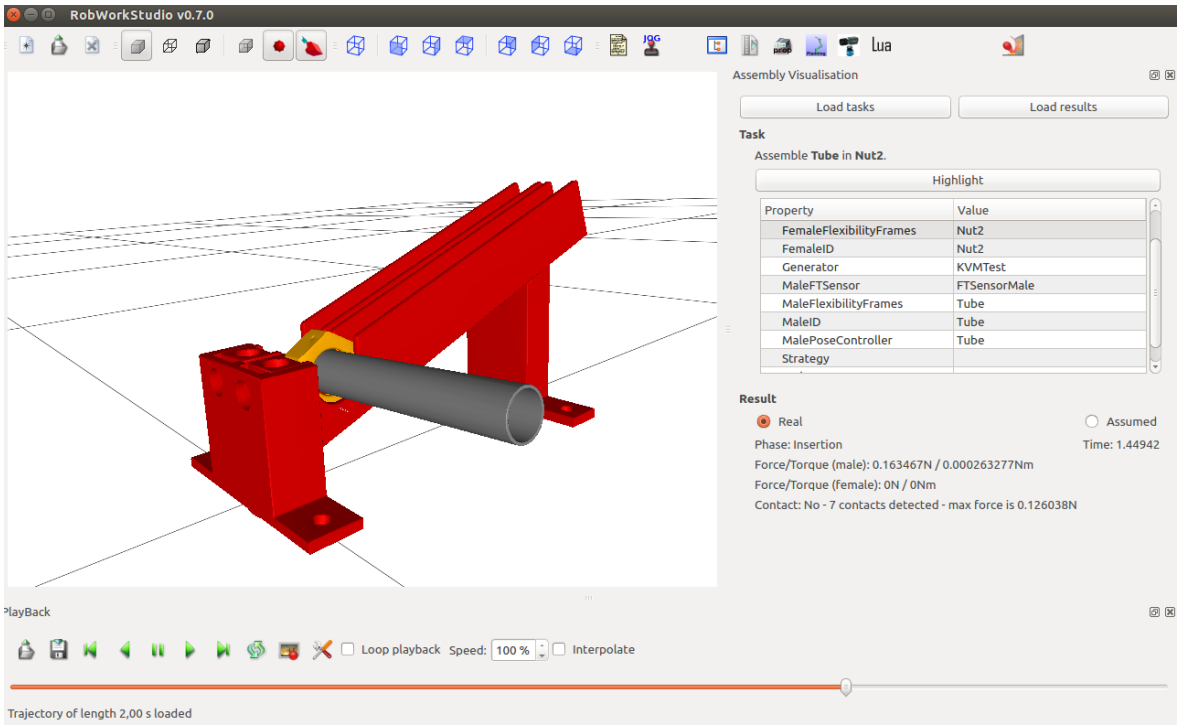


Figure 7.2: The assembly visualisation plugin for RobWorkStudio

work for assembly and the clear distinction between the real state and assumed knowledge will facilitate the development of algorithms for robust assembly. The framework provides

visualisation that makes it easy to look into the control strategy and compare the simulated "reality" with the controllers view of the world. Furthermore it defines an assembly action as a relative motion on an abstract level. This will allow user-friendly definition of complex behaviours by chaining of atomic actions. By defining the action as a relative motion we make the least possible assumptions about the system the strategy is used in.

CHAPTER 8

Conclusion

A new approach to dynamic simulation is presented in this dissertation. Here, the used methods are based on optimisation of certain objective functions instead of the often used LCP formulations. The objectives incorporate the desired characteristic of distributed loads, which we find are important when we focus on the force space results of dynamic simulation. In manipulation, there will often be redundancy in the dynamics, and we must rely on stable force space results for doing reliable control strategies. Furthermore, the new engine allows modelling of the interaction between bodies. For colliding contacts, the classical restitution is used while a micro-slip friction model is used to model friction.

The engine is evaluated in some tests. The results illustrate that the engine has some of the desired features in connection with simulation of assembly actions. The engine maintains sufficient computational speed while at the same time avoiding penetrations completely. Hence, it can handle tight-fitting scenarios.

A special approach to parallelisation of work in multiple threads is introduced. This facilitates optimum usage of the resources allocated for simulation. Dynamic engines are known to be difficult to debug, and this engine is no exception. For this reason, we provide extended graphical debugging capabilities allowing the user to follow the internal computations step by step. We estimate this a very important feature. The framework can be extended by providing plugins, allowing the user to implement certain customisations.

A new framework for assembly is introduced in chapter 7. This framework provides a unified way to specify and simulate assembly actions. The goal is to provide an environment for development of control strategies, where no distinction between a real and simulated environment is made. This unified framework for assembly simulation also allows the best strategies and optimisation of these strategies in simulation. For development of control strategies, an interface for MatLab is created. This allows the users, who are more familiar with MatLab, to use our simulation tools for development of controllers.

Overall, a new dynamic simulator, tailored to manipulation and assembly, is developed. It overcomes some of the difficulties in the area and facilitates development of robust control strategies in simulation.

CHAPTER 9

Future Work

The developed engine poses some interesting challenges, which we find should be further investigated. The main difficulty in the simulation method, is reliable tracking of contacts. We find that future work should focus on the contact detection aspects, ensuring that contacts move smoothly on the surface of objects during simulation.

In regards of important software features, the following are examples, which we find important to develop further:

- Integration with other frameworks used in robotics for dynamic simulation, such as Gazebo and the Physics Abstraction Layer (PAL).
- Increased user friendliness with graphical interfaces and guidance for choosing parameters. In addition, intuitive specification of assembly strategies and parameters are needed.

In relation to validation of the engine, more tests are needed. We propose deeper testing of the following features, which has not been in focus in this dissertation:

- Handling of large systems of colliding bodies. Until now, focus has been on assembly actions, where the collision can typically be handled in a chain-like fashion. The hybrid collision solver should, however, be tested in a more general setting.
- Constraints should be tested in greater depth. We have focused on modelling of control through modelling of compliance. In general, the combination of bilateral and unilateral constraints should be considered in more detail. For instance for use of articulated grippers. In the iterative methods, focus has been on combining different objectives for bilateral and unilateral constraints. For this reason, future work will mostly be centered on choice of correct parameters for the objective functions.
- Stacking tests are not necessarily important in the types of test we propose in this dissertation. We do expect that these are handled robustly, but future tests will show.

In general, we find that the engine introduces many numerical parameters, for instance for thresholds, iteration limits and target precisions. Determining an optimum set of default parameters is ongoing work.

Bibliography

[1] T. N. Thulesen, "Dynamic Grasp Simulation in RobWork," Master's thesis, University of Southern Denmark, June 2012.

[2] L.-P. Ellekilde and J. A. Jorgensen, "RobWork: A Flexible Toolbox for Robotics Research and Education," *Robotics (ISR), 2010 41st International Symposium on and 2010 6th German Conference on Robotics (ROBOTIK)*, pp. 1 –7, june 2010.

[3] T. N. Thulesen and H. G. Petersen, "RobWorkPhysicsEngine: A new Dynamic Simulation Engine for Manipulation Actions," in *Robotics and Automation (ICRA), 2016 IEEE International Conference on*, 2016 (submitted).

[4] ABB RobotStudio. [Accessed Nov. 14, 2015]. [Online]. Available: http://new.abb.com/products/robotics/robotstudio

[5] I. Adept Technology. Adept ace - robot control software. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.adept.com/products/software/pc/adept-ace/general

[6] D. Robotics. Wincaps iii. [Accessed Nov. 14, 2015]. [Online]. Available: http://densorobotics.com/products/off-line-software-wincaps

[7] FANUC. Roboguide. [Accessed Nov. 14, 2015]. [Online]. Available: http://robot.fanucamerica.com/products/vision-software/ROBOGUIDE-simulation-software.aspx

[8] K. Robotics. Simulation and olp software for kawasaki robots. [Accessed Nov. 14, 2015]. [Online]. Available: https://robotics.kawasaki.com/en/products/other/simulation-OLP

[9] K. I. Robots. Kuka.sim. [Accessed Nov. 14, 2015]. [Online]. Available: http://kukasim.com

[10] V. Components. 3d factory simulation products. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.visualcomponents.com/products

[11] E. Robotics and M. Vision. Actin simulation. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.energid.com/software/actin-simulation

[12] RoboDK. Robot development kit for offline programming. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.robodk.com

[13] anyKode. Marilou - modeling and simulation environment for robotics. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.anykode.com

[14] Robologix. [Accessed Nov. 14, 2015]. [Online]. Available: https://www.robologix.com

[15] A. Robotics. Artiminds rps. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.artiminds.com

[16] Robotic simulator: Roboworks. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.newtonium.com

[17] D. Systèmes. Delmia. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.3ds.com/products-services/delmia

[18] O. Michel, "Webots: Professional Mobile Robot Simulation," *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, pp. 39–42, 2004.

[19] M. Software. Adams. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.mscsoftware.com/product/adams

[20] IT+Robotics. Workcellsimulator. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.it-robotics.it/products/3d-simulation/workcellsimulator

[21] Microsoft. Microsoft robotics developer studio 4. [Accessed Nov. 14, 2015]. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=29081

[22] C. Pinciroli, V. Trianni, R. O'Grady, G. Pini, A. Brutschy, M. Brambilla, N. Mathews, E. Ferrante, G. Di Caro, F. Ducatelle, T. Stirling, A. Gutierrez, L. Gambardella, and M. Dorigo, "ARGoS: A modular, multi-engine simulator for heterogeneous swarm robotics," in *Intelligent Robots and Systems (IROS), 2011 IEEE/RSJ International Conference on*, Sept 2011, pp. 5027–5034.

[23] ARS: Python robotics simulator. [Accessed Nov. 19, 2015]. [Online]. Available: https://bitbucket.org/glarrain/ars

[24] A. Shapiro, P. Faloutsos, and V. Ng-Thow-Hing, "Dynamic animation and control environment," in *Proceedings of Graphics Interface 2005*, ser. GI '05. School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada: Canadian Human-Computer Communications Society, 2005, pp. 61–70. [Online]. Available: http://dl.acm.org/citation.cfm?id=1089508.1089519

[25] N. Koenig and A. Howard, "Design and use paradigms for Gazebo, an open-source multi-robot simulator," in *Intelligent Robots and Systems, 2004. (IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, vol. 3, Sept 2004, pp. 2149–2154 vol.3.

[26] B. León, S. Ulbrich, R. Diankov, G. Puche, M. Przybylski, A. Morales, T. Asfour, S. Moisio, J. Bohg, J. Kuffner, and R. Dillmann, "OpenGRASP: A Toolkit for Robot Grasping Simulation," in *Simulation, Modeling, and Programming for Autonomous Robots*. Springer Berlin Heidelberg, 2010, pp. 109–120. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17319-6_13

[27] L. Beatriz, G. Puche, H. Martí, A. Morales, S. Ulbrich, T. Asfour, R. Diankov, J. Kuffner, and S. Moisio, "OpenGRASP: A New Robot Grasping Simulation Toolkit," 2011.

[28] F. Kanehiro, K. Fujiwara, S. Kajita, K. Yokoi, K. Kaneko, H. Hirukawa, Y. Nakamura, and K. Yamane, "Open architecture humanoid robotics platform," in *Robotics and Automation, 2002. Proceedings. ICRA '02. IEEE International Conference on*, vol. 1, 2002, pp. 24–30 vol.1.

[29] R. Diankov, "Automated Construction of Robotic Manipulation Programs," Ph.D. dissertation, Carnegie Mellon University, Robotics Institute, August 2010. [Online]. Available: http://www.programmingvision.com/rosen_diankov_thesis.pdf

[30] S. L. Delp, F. C. Anderson, A. S. Arnold, P. Loan, A. Habib, T. John, E. Guendelman, and D. G. Thelen, "OpenSim: Open-source Software to Create and Analyze Dynamic Simulations of Movement."

[31] G. Echeverria, N. Lassabe, A. Degroote, and S. Lemaignan, "Modular OpenRobots Simulation Engine: MORSE," in *Proceedings of the IEEE ICRA*, 2011.

[32] O. Obst and M. Rollmann, "Spark – A Generic Simulator for Physical Multi-Agent Simulations," in *Computer Systems Science and Engineering*, 2004.

[33] M. F. E. Rohmer, S. P. N. Singh, "V-REP: a Versatile and Scalable Robot Simulation Framework," in *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*, 2013.

[34] ROS: Robot Operating System. [Accessed Nov. 19, 2015]. [Online]. Available: http://www.ros.org

[35] S. Ivaldi, J. Peters, V. Padois, and F. Nori, "Tools for simulating humanoid robot dynamics: A survey based on user feedback," in *Humanoid Robots (Humanoids), 2014 14th IEEE-RAS International Conference on*, Nov 2014, pp. 842–849.

[36] A. Boeing and T. Bräunl, "Evaluation of Real-time Physics Simulation Systems," in *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, ser. GRAPHITE '07. New York, NY, USA: ACM, 2007, pp. 281–288. [Online]. Available: http://doi.acm.org/10.1145/1321261.1321312

[37] A. Boeing, "Design of a Physics Abstraction Layer for Improving the Validity of Evolved Robot Control Simulations," Ph.D. dissertation, The University of Western Australia, May 2009. [Online]. Available: http://robotics.ee.uwa.edu.au/theses/2009-PhysicsSimulation-Boeing-PhD.pdf

[38] OPAL: Open Physics Abstraction Layer. [Accessed Nov. 13, 2015]. [Online]. Available: http://opal.sourceforge.net

[39] J. A. Jørgensen, L.-P. Ellekilde, and H. G. Petersen, "RobWorkSim - an Open Simulator for Sensor based Grasping," in *ISR/ROBOTIK 2010 - ISR 2010 (41st International Symposium on Robotics) and ROBOTIK 2010 (6th German Conference on Robotics)*. VDE-Verlag, June 2010. [Online]. Available: http://www.vde-verlag.de/proceedings-en/453273198.html

[40] E. Coumans and K. Victor, "COLLADA Physics," in *Proceedings of the Twelfth International Conference on 3D Web Technology*, ser. Web3D '07. New York, NY, USA: ACM, 2007, pp. 101–104. [Online]. Available: http://doi.acm.org/10.1145/1229390.1229407

[41] K. Erleben, "Stable, Robust, and Versatile Multibody Dynamics Animation," Ph.D. dissertation, University of Copenhagen, Denmark, 2005.

[42] H. Garstenauer, "A unified framework for rigid body dynamics," Master's thesis, Johannes Kepler Universität, Linz, March 2006.

[43] J. Bender, "Impulsbasierte Dynamiksimulation von Mehrkörpersystemen in der virtuellen Realität," Ph.D. dissertation, Universität Karlsruhe, 2007. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/documents/2563

[44] D. Pogorelov, "Differential–algebraic equations in multibody system modeling," *Numerical algorithms*, vol. 19, no. 1-4, pp. 183–194, 1998.

[45] C. Bendtsen and P. G. Thomsen, "Numerical Solution of Differential Algebraic Equations," Tech. Rep., May 1999.

[46] M. G. Hollars, D. E. Rosenthal, and M. A. Sherman, "SD/FAST User's Manual," *Symbolic Dynamics Inc., Mountain View, CA, USA*, September 1994.

[47] D. Baraff, "Analytical Methods for Dynamic Simulation of Non-penetrating Rigid Bodies," in *ACM SIGGRAPH Computer Graphics*, vol. 23, no. 3. ACM, 1989, pp. 223–232.

[48] B. Nguyen and J. C. Trinkle, "dvc3D: a three dimensional physical simulation tool for rigid bodies with contacts and coulomb friction," in *The 1st Joint International Conference on Multibody System Dynamics*, 2010.

[49] D. Baraff, "Coping with Friction for Non-penetrating Rigid Body Simulation," in *Proceedings of the 18th Annual Conference on Computer Graphics and Interactive Techniques*, ser. SIGGRAPH '91. New York, NY, USA: ACM, 1991, pp. 31–41. [Online]. Available: http://doi.acm.org/10.1145/122718.122722

[50] J. K. Hahn, "Realistic Animation of Rigid Bodies," in *ACM SIGGRAPH Computer Graphics*, vol. 22, no. 4. ACM, 1988, pp. 299–308.

[51] B. V. Mirtich, "Impulse-based dynamic simulation of rigid body systems," Ph.D. dissertation, University of California at Berkeley, 1996.

[52] D. E. Stewart and J. C. Trinkle, "An implicit time-stepping scheme for rigid body dynamics with inelastic collisions and coulomb friction," *International Journal for Numerical Methods in Engineering*, vol. 39, no. 15, pp. 2673–2691, 1996. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1097-0207(19960815)39:15<2673::AID-NME972>3.0.CO;2-I

[53] D. Baraff, "Fast contact force computation for nonpenetrating rigid bodies," in *Proceedings of the 21st annual conference on Computer graphics and interactive techniques*. ACM, 1994, pp. 23–34.

[54] M. Anitescu and F. A. Potra, "Formulating Dynamic Multi-Rigid-Body Contact Problems with Friction as Solvable Linear Complementarity Problems," *Nonlinear Dynamics*, vol. 14, no. 3, pp. 231–247, 1997.

[55] M. Anitescu, F. A. Potra, *et al.*, "A time-stepping method for stiff multibody dynamics with contact and friction," *International journal for numerical methods in engineering*, vol. 55, no. 7, pp. 753–784, 2002.

[56] S. Reich, "Stabilization of Constrained Mechanical Systems with DAEs and Invariant Manifolds," 1993.

[57] E. Coumans, *Bullet 2.83 Physics SDK Manual*, 2015.

[58] P. Masarati, M. Morandini, and P. Mantegazza, "An Efficient Formulation for General-Purpose Multibody/Multiphysics Analysis," *Journal of Computational and Nonlinear Dynamics*, vol. 9, no. 4, p. 041001, 2014.

[59] RBDL: Rigid Body Dynamics Library. [Accessed Nov. 14, 2015]. [Online]. Available: http://rbdl.bitbucket.org

[60] DART: Dynamic Animation and Robotics Toolkit. [Accessed Nov. 13, 2015]. [Online]. Available: http://dartsim.github.io

[61] Moby. [Accessed Nov. 14, 2015]. [Online]. Available: http://sourceforge.net/projects/physsim

[62] DynaMechs. [Accessed Nov. 14, 2015]. [Online]. Available: http://sourceforge.net/projects/dynamechs

[63] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*, Oct 2012, pp. 5026–5033.

[64] V. Acary and F. Pérignon, "An introduction to Siconos," INRIA, Tech. Rep., July 2007.

[65] Newton Game Dynamics. [Accessed Nov. 14, 2015]. [Online]. Available: http://newtondynamics.com

[66] M. A. Sherman, A. Seth, and S. L. Delp, "Simbody: multibody dynamics for biomedical research," *Procedia {IUTAM}*, vol. 2, pp. 241 – 261, 2011, {IUTAM} Symposium on Human Body Dynamics. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S2210983811000241

[67] IBDS. [Accessed Nov. 14, 2015]. [Online]. Available: http://sourceforge.net/projects/ibds

[68] R. Smith. Open Dynamics Engine. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.ode.org

[69] F. Faure, C. Duriez, H. Delingette, J. Allard, B. Gilles, S. Marchesseau, H. Talbot, H. Courtecuisse, G. Bousquet, I. Peterlik, *et al.*, "Sofa: A multi-model framework for interactive physical simulation," in *Soft Tissue Biomechanical Modeling for Computer Assisted Surgery*. Springer, 2012, pp. 283–321.

[70] JigLib - Rigid body physics engine. [Accessed Nov. 14, 2015]. [Online]. Available: http://www.rowlhouse.co.uk/jiglib

[71] T. Koziara and N. Bicanic, "On Large Scale Implicit Multibody Contact Dynamics Modeling with SOLFEC," in *Modelling and Measuring Reactor Core Graphite Properties and Performance*. The Royal Society of Chemistry, 2013, pp. 84–90. [Online]. Available: http://dx.doi.org/10.1039/9781849735179-00084

[72] OpenTissue. [Accessed Nov. 13, 2015]. [Online]. Available: http://www.opentissue.org

[73] MBSim. [Accessed Nov. 14, 2015]. [Online]. Available: https://github.com/mbsim-env/mbsim

[74] C. Deul, P. Charrier, and J. Bender, "Position-based rigid-body dynamics," *Computer Animation and Virtual Worlds*, pp. n/a–n/a, 2014. [Online]. Available: http://dx.doi.org/10.1002/cav.1614

[75] M. Tamis, "Comparison between Projected Gauss Seidel and Sequential Impulse Solvers for Real-Time Physics Simulations," July 2015.

[76] D. E. Stewart, "Rigid-Body Dynamics with Friction and Impact," *SIAM Review*, vol. 42, no. 1, pp. 3–39, 2000.

[77] S. McMillan, "DynaMechs Simulation Library – reference Maual," Tech. Rep., 1995.

[78] E. Guendelman, R. Bridson, and R. Fedkiw, "Nonconvex Rigid Bodies with Stacking," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 871–878, July 2003. [Online]. Available: http://doi.acm.org/10.1145/882262.882358

[79] A. Seugling and M. Rölin, "Evaluation of Physics Engines and Implementation of a Physics Module in a 3d-Authoring Tool," Master's thesis, Umeå University, Sweden, March 2006.

[80] A. Boeing and T. Bräunl, "Evaluation of Real-time Physics Simulation Systems," in *Proceedings of the 5th International Conference on Computer Graphics and Interactive Techniques in Australia and Southeast Asia*, ser. GRAPHITE '07. New York, NY, USA: ACM, 2007, pp. 281–288. [Online]. Available: http://doi.acm.org/10.1145/1321261.1321312

[81] T. Erez, Y. Tassa, and E. Todorov, "Simulation Tools for Model-Based Robotics: Comparison of Bullet, Havok, MuJoCo, ODE and PhysX," *International Conference on Robotics and Automation*, 2015.

[82] Assimp: Open Asset Import Library. [Accessed Nov. 18, 2015]. [Online]. Available: http://assimp.sourceforge.net

[83] R. McNeel. openNURBS SDK. [Accessed Nov. 18, 2015]. [Online]. Available: https://www.rhino3d.com/opennurbs

[84] B. Mirtich, "Fast and Accurate Computation of Polyhedral Mass Properties," *J. Graph. Tools*, vol. 1, no. 2, pp. 31–50, Feb. 1996. [Online]. Available: http://dx.doi.org/10.1080/10867651.1996.10487458

[85] U. o. N. C. Department of Computer Science. Collision Detection and Proximity Query Packages. [Accessed Nov. 18, 2015]. [Online]. Available: http://gamma.cs.unc.edu/research/collision/packages.html

[86] S. Gottschalk, M. C. Lin, and D. Manocha, "OBBTree: A Hierarchical Structure for Rapid Interference Detection," in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*. ACM, 1996, pp. 171–180.

[87] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha, "Fast Proximity Queries with Swept Sphere Volumes," Technical Report TR99-018, Department of Computer Science, University of North Carolina, Tech. Rep., 1999.

[88] E. Gilbert, D. Johnson, and S. Keerthi, "A fast procedure for computing the distance between complex objects in three-dimensional space," *Robotics and Automation, IEEE Journal of*, vol. 4, no. 2, pp. 193–203, Apr 1988.

[89] K. Hauser, "Robust contact generation for robot simulation with unstructured meshes," *International Symposium on Robotics Research, Singapore*, Dec. 2013.

[90] K. Guo, X. Zhang, H. Li, H. Hua, and G. Meng, "A New Dynamical Friction Model," *International Journal of Modern Physics B*, vol. 22, no. 08, pp. 967–980, 2008. [Online]. Available: http://www.worldscientific.com/doi/abs/10.1142/S0217979208039010

[91] R. Smith. (2006, February) Open Dynamics Engine v0.5 User Guide. [Accessed Nov. 25, 2015]. [Online]. Available: http://www.ode.org/ode-latest-userguide.html

[92] Simplified Wrapper and Interface Generator. [Accessed Nov. 25, 2015]. [Online]. Available: http://www.swig.org

[93] (2015) RobWork API Documentation. [Online]. Available: http://www.robwork.dk/apidoc/nightly/rw

[94] Wolfram. (2015) WSTP: Wolfram Symbolic Transfer Protocol. [Online]. Available: https://www.wolfram.com/wstp

[95] A.-u.-R. Mustafa, "Parallelization of the Webots simulation loop & the ODE physics engine," Master's thesis, École Polytechnique Fédérale de Lausanne, 2014.

[96] E. Drumwright, J. Hsu, N. Koenig, and D. Shell, "Extending Open Dynamics Engine for Robotics Simulation," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, vol. 6472, pp. 38–50. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-17319-6_7

[97] Open Source Robotics Foundation, "DARPA Robotics Challenge OSRF Gazebo Parallel Physics Report," Tech. Rep., May 2015.

[98] J. Buch, J. Laursen, L. Sørensen, L.-P. Ellekilde, D. Kraft, U. Schultz, and H. Petersen, "Applying Simulation and a Domain-Specific Language for an Adaptive Action Library," in *Simulation, Modeling, and Programming for Autonomous Robots*, ser. Lecture Notes in Computer Science, D. Brugali, J. Broenink, T. Kroeger, and B. MacDonald, Eds. Springer International Publishing, 2014, vol. 8810, pp. 86–97. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-11900-7_8